

Object Oriented Programming



Syllabus

OBJECT ORIENTED PROGRAMMING C++

1. Introduction :

What is object oriented programming? Why do we need object-oriented. Programming characteristics of object-oriented languages. C and C++.

2. C++ Programming basics :

Output using cout. Directives. Input with cin. Type bool. The setw manipulator. Type conversions.

3. Functions :

Returning values from functions. Reference arguments. Overloaded function. Inline function. Default arguments. Returning by reference.

4. Object and Classes :

Making sense of core object concepts (Encapsulation, Abstraction, Polymorphism, Classes, Messages Association, Interfaces) Implementation of class in C++, C++ Objects as physical object, C++ object as data types constructor. Object as function arguments. The default copy constructor, returning object from function. Structures and classes. Classes objects and memory static class data. Const and classes.

5. Arrays and string arrays fundamentals. Arrays as class Member Data :

Arrays of object, string, The standard C++ String class

6. Operator overloading :

Overloading unary operations. Overloading binary operators, data conversion, pitfalls of operators overloading and conversion keywords. Explicit and Mutable.

7. Inheritance :

Concept of inheritance. Derived class and based class. Derived class constructors, member function, inheritance in the English distance class, class hierarchies, inheritance and graphics shapes, public and private inheritance, aggregation : Classes within classes, inheritance and program development.

8. Pointer :

Addresses and pointers. The address of operator and pointer and arrays. Pointer and Fraction pointer and C-types string. Memory management : New and Delete, pointers to objects, debugging pointers.

9. Virtual Function :

Virtual Function, friend function, Static function, Assignment and copy initialization, this pointer, dynamic type information.

10. Streams and Files :

Streams classes, Stream Errors, Disk File I/O with streams, file pointers, error handling in file I/O with member function, overloading the extraction and insertion operators, memory as a stream object, command line arguments, and printer output.

11. Templates and Exceptions :

Function templates, Class templates Exceptions

12. The Standard Template Library :

Introduction algorithms, sequence containers, iterators, specialized iterators, associative containers, strong user-defined object, function objects.

Term work / Practical : Each candidate will submit a journal in which at least 10 assignments based on the above syllabus and the internal paper. Test will be graded for 10 marks and assignments will be graded for 15 marks.

1. Object Oriented Programming in C++ by Robert Lafore Techmedia Publication.
2. The complete reference C – by Herbert shieldt Tata McGraw Hill Publication.
3. Object Oriented Programming in C++ Saurav Sahay Oxford University Press.
4. Object Oriented Programming in C++ R Rajaram New Age International Publishers 2nd.
5. OOPS C++ Big C++ Cay Horstmann Wiley Publication.

Practical for C++

Programming exercises and project using C++ programming languages, to study various features of the languages. Stress to be laid on writing well structured modular and readable programs accompanied by good documentation.

The topic wise assignments are as follows :

1. Function Blocks

- a. Handling default reference arguments
- b. Handling inline and overloaded function

2. Objects and Classes

- a. Creating UDT using classes and object

3. Arrays and String as objects

- a. Insertion, Deletion, reversal sorting of elements into a single



FUNDAMENTALS OF C++

Unit Structure

1. Introduction of C++
2. Object-Oriented Programming
3. Encapsulation
4. Polymorphism
5. Inheritance
6. The Need for C++
7. Characteristics of OOPs
8. C++ and C

1.1 INTRODUCTION OF C++

The history of C++ begins with C. C++ is built upon the foundation of C. Thus, C++ is a superset of C. C++ expanded and enhanced the C language to support object-oriented programming (which is described later in this module). C++ also added several other improvements to the C language, including an extended set of library routines. However, much of the spirit and flavor of C++ is directly inherited from C. Therefore, to fully understand and appreciate C++, you need to understand the “how and why” behind C.

C++ was invented by Bjarne Stroustrup in 1979, at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language “C with Classes ” However in 1983 the name was changed to C++ Stroustrup built C++ on the foundation of C including all of C’s features attributes and benefits He also adhered to C’s underlying philosophy that the programmer not the language is in charge t this point it is critical to understand that Stroustrup did not create an entirely new programming language. Instead, he enhanced an already highly successful language.

Most of the features that Stroustrup added to C were designed to support object-oriented programming. In essence, C++ is the object-oriented version of C. By building upon the foundation of C, Stroustrup provided a smooth migration path to OOP. Instead of having to learn an entirely new language, a C programmer needed to learn only a few new features before reaping the benefits of the object-oriented methodology

2. OBJECT-ORIENTED PROGRAMMING

Central to C++ is object-oriented programming (OOP). OOP was the impetus for the creation of C++. Because of this it is useful to understand OOP's basic principles before you write even a simple C++ program.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different and better way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data"

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code" In an object-oriented language; you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages, including C++, have three traits in common: encapsulation, polymorphism and inheritance. Let's examine each.

3. ENCAPSULATION

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained black box is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

4. POLYMORPHISM

Polymorphism (from Greek meaning "many forms") is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus,

turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.

The same principle can also apply to programming. For example, consider a stack (which is a first-in, last-out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in C++ you can create one general set of stack routines that works for all three situations. This way, once you know how to use one stack, you can use them all.

5. INHERITANCE

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. That is, the food class possesses certain qualities (edible, nutritious, and so on) which also, logically, apply to its subclass, fruit. In addition to these qualities, the fruit class has specific characteristics (juicy, sweet, and so on) that distinguish it from other food. The apple class defines those qualities specific to an apple (grows on trees, not tropical, and so on). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes and would define only those qualities that make it unique.

Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

6. THE NEED FOR C++

Given the preceding discussion, you might be wondering why C++ was invented. Since C was a successful computer programming language, why was there a need for something else?

The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand the correlation between increasing program complexity and computer language development, consider the following. Approaches to programming have changed dramatically since the invention of the computer.

For example when computers were first invented programming was done by using the computer's front panel to toggle in the binary machine instructions. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that programmers could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were developed to give programmers more tools with which to handle the complexity.

The first widely used computer language was, of course, FORTRAN. While FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to-understand programs. The 1960s gave birth to structured programming, which is the method of programming encouraged by languages such as C. With structured languages it was, for the first time, possible to write moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the late 1970s, many projects were near or at this point.

In response to this problem, a new way to program began to emerge: object-oriented programming (OOP). Using OOP, a programmer could handle larger, more complex programs. The trouble was that C did not support object-oriented programming. The desire for an object-oriented version of C ultimately led to the creation of C++.

In the final analysis, although C is one of the most liked and widely used professional programming languages in the world, there comes a time when its ability to handle complexity is exceeded. Once a program reaches a certain size, it becomes so complex that it is difficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken and to help the programmer comprehend and manage larger, more complex programs.

1.7 CHARACTERISTICS OF OOPS

OOP offers several benefits to both the program designer and the user. The principal advantages are.

- 1) Emphasis is on data rather than procedure.
- 2) Programs are divided into what are known as objects.
- 3) Data structures are designed such that they characterize the objects.
- 4) Functions that operate on the data of an object are tied together in the data structure.
- 5) Data is hidden and cannot be accessed by external functions.
- 6) Objects may communicate with each other through functions.
- 7) New data and functions can be easily added wherever necessary.
- 8) Follows bottom up approach in program design.
- 9) Through inheritance, we can eliminate redundant code and extend the use of existing classes
- 10) We can build program from the standard working module that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- 11) The principal of data hiding helps the programmer to build secure programs that cannot be invaded by code in other part of the program.
- 12) It is possible to have multiple instance of an object to co-exist without any interference
- 13) It is easy to partition the work in a project, based on objects.
- 14) Object oriented systems can be easily upgraded from small to large systems.
- 15) Message passing techniques for communication between objects makes the interface description with external systems much simpler.
- 16) Software complexity can be easily managed.

8. C++ AND C

- C is procedure oriented programming & c++ is object oriented programming.
- C is top to bottom programming approach; c++ is bottom to top programming approach.

- C ++ has inheritance concept but c has not inheritance concept
- C is structured design, c++ is object oriented design
- C input/output is based on library and the processes are carried out by including functions.
- C++ I/O is made through console commands cin and cout.
- Undeclared functions in c++ are not allowed. The function has to have a prototype defined before the main() before use in c++ although in c the functions can be declared at the point of use.
- Although it is possible to implement anything which C++ could implement in C, C++ aids to standardize a way in which objects are created and managed, whereas the C programmer who implements the same system has a lot of liberty on how to actually implement the internals, and style among programmers will vary a lot on the design choices made.



C++ PROGRAMMING BASICS

Unit Structure

1. Some of the important concepts in oops are
2. Basic Data Types in C++
3. Variables in C++
4. Operators in C++

2.1 SOME OF THE IMPORTANT CONCEPTS IN OOPS ARE

- 1) Objects
- 2) Classes
- 3) Data abstraction & Encapsulation.
- 4) Inheritance
- 5) Polymorphism
- 6) Dynamic binding.
- 7) Message passing.

2.1.1 Object:-

- i) Objects are the basic run-time entities in an object-oriented system.
- ii) They may represent a person, a place a bank account, a table of data or any item that the program has to handle.
- iii) Programming problem is analyzed in terms of objects and the nature of communication between them.
- iv) Objects take up space in the memory & have an associated address like structure in c.
- v) When a program executes, the object interacts by sending messages to one another.

Ex. If there are two objects “customer” and “account”, then the customer object may send a message to the account object requesting for the bank balance. Thus each object contains data, and code to manipulate the data.

2.1.2 Classes

- i) The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are actually variable of the type class.
- ii) Once a class has been defined, we can create any number of objects belonging to that class. Thus a class is collection of objects of similar type.
- iii) Classes are user defined data types and behaves like the built in type of a programming language.
- iv) The syntax for defining class is class class-name

```
{
-----
-----
}
```

2.1.3 Data abstraction and Encapsulation

- i) The wrapping up of data and functions into a single unit called class is known as encapsulation.
- ii) The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- iii) These functions provide the interface between the objects data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.
- iv) Abstraction refers to the act of representing essential features without including the background details or explanations.
- v) Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and coast, and functions to operate on these attributes.

2.1.4 Inheritance

- i) Inheritance is the process by which object of one class acquire the properties of objects of another class.
- ii) In OOPs, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.
- iii) This is possible by deriving a new class from the existing one. The new class will have combined features of both the classes.

2.1.5 Polymorphism

- i) Polymorphism is important oops concept. It means ability to take more than one form.

- ii) In polymorphism an operation may show different behavior in different instances. The behavior depends upon the type of data used in the operation.

For Ex- Operation of addition for two numbers, will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

- iii) The process of making an operator to show different behavior in different instance is called as operator overloading. C++ support operator overloading.

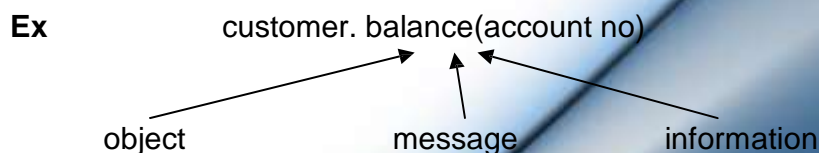
2.1.6 Dynamic Binding

- i) Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- ii) Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.

2.1.7 Message Passing

- i) OOPs consist of a set of objects that communicate with each other.
- ii) Message passing involves following steps
- iii) Creating classes that define objects and their behavior
- iv) Creating objects from class definitions and
- v) Establishing communication among objects.
- vi) A message for an object is a request for execution of a procedure & therefore will invoke a function in the receiving object that generates the desired result.

Message passing involves specifying the name of the object, the name of the function i.e. message and the information to be sent.



A First Simple C++ Program

```
#include<iostream.h>
int main()
{
    cout<<".Hello World.";
    return 0;
}
```

- C++ program is a collection of functions. Every C++ program must have a main() function.
- The iostream file:-
The header file iostream should be included at the beginning of all programs that uses one output statement.
- Input / Output operator

1. Input Operator cin :-

The identifier cin is a predefined object in c++ that corresponds to the standard input stream. Here this stream represents keyboard.

Syntax:- cin>>variable;

The operator >> is known as extraction or get from operator & assigns it to the variable on its right.

2. Output operator cout :-

The identifier cout is predefined object that represents the standard output stream in c++. Here standard output stream represents the screen.

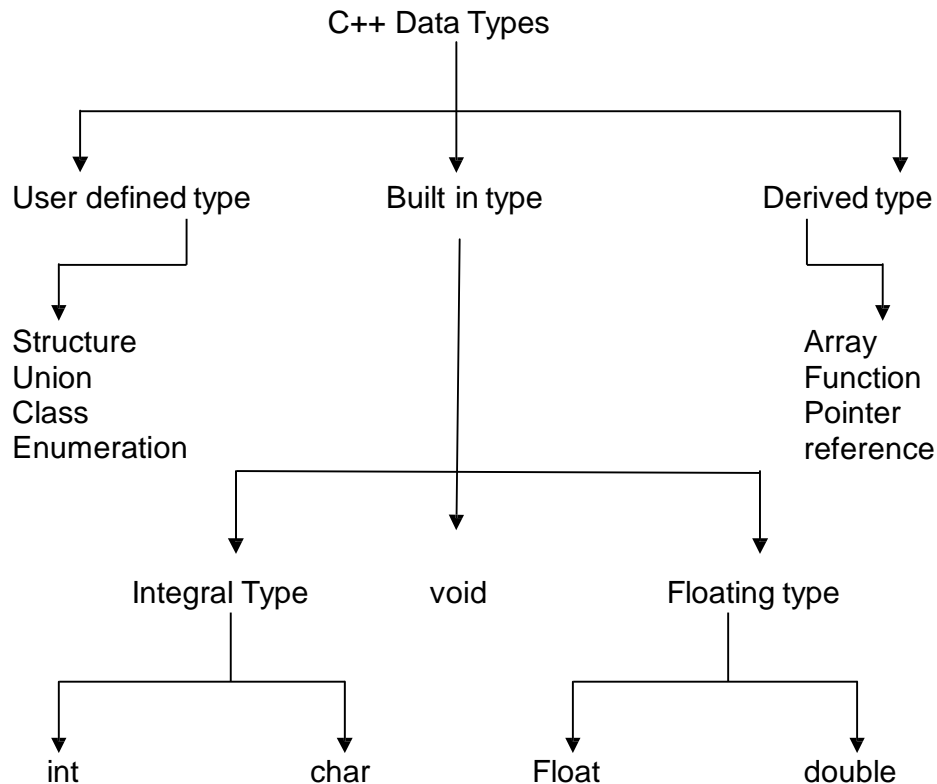
Syntax:- cout<<string;

The operator << is called the insertion or put to operator. It inserts the contents of the variable on its right to the object on its left.

3. Return type of main():-

In C++, main returns an integer type value to the operating system. So return type for main() is explicitly specified as int. Therefore every main() in c++ should end with a return 0 statement.

2.2 BASIC DATA TYPES IN C++



Built-in-type

1) Integral type : – The data types in this type are int and char. The modifiers signed, unsigned, long & short may be applied to character & integer basic data type. The size of int is 2 bytes and char is 1 byte.

2) void – void is used

- i) To specify the return type of a function when it is not returning any value.
- ii) To indicate an empty argument list to a function.

Ex- Void function1(void)

- iii) In the declaration of generic pointers.

EX- void *gp

A generic pointer can be assigned a pointer value of any basic data type.

Ex . `int *ip // this is int pointer`
 `gp = ip //assign int pointer to void.`

A void pointer cannot be directly assigned to their type pointers in c++ we need to use cast operator.

Ex - `Void * ptr1;`
 `Char * ptr2;`
 `ptr2 = ptr1;`
 is allowed in c but not in c++

`ptr2 = (char *)ptr1`
 is the correct statement

3) Floating type:

The data types in this are float & double. The size of the float is 4 byte and double is 8 byte. The modifier long can be applied to double & the size of long double is 10 byte.

User-defined type:

- i) The user-defined data type **structure** and **union** are same as that of C.
- ii) **Classes** – Class is a user defined data type which can be used just like any other basic data type once declared. The class variables are known as objects.

iii) Enumeration

- a) An enumerated data type is another user defined type which provides a way of attaching names to numbers to increase simplicity of the code.

b) It uses enum keyword which automatically enumerates a list of words by assigning them values 0, 1, 2,...etc.

c) Syntax:-

```
enum shape {
    circle,
    square,
    triangle
}
```

Now shape becomes a new type name & we can declare new variables of this type.

EX . `shape oval;`

- d) In C++, enumerated data type has its own separate type. Therefore c++ does not permit an int value to be automatically converted to an enum value.

Ex. `shape shapes1 = triangle, // is allowed`
 `shape shape1 = 2; // Error in c++`
 `shape shape1 = (shape)2; //ok`

- e) By default, enumerators are assigned integer values starting with 0, but we can over-ride the default value by assigning some other value.

EX `enum colour {red, blue, pink = 3}; //it will assign red to 0, blue to 1, & pink to 3` or `enum colour {red = 5, blue, green}; //it will assign red to 5, blue to 6 & green to 7.`

Derived Data types:

1) Arrays

An array in c++ is similar to that in c, the only difference is the way character arrays are initialized. In c++, the size should be one larger than the number of character in the string where in c, it is exact same as the length of string constant.

Ex - `char string1[3] = "ab"; // in c++`
 `char string1[2] = "ab"; // in c.`

2) Functions

Functions in c++ are different than in c there is lots of modification in functions in c++ due to object orientated concepts in c++.

3) Pointers

Pointers are declared & initialized as in c.

Ex `int * ip; // int pointer`
 `ip = &x; //address of x through indirection`

c++ adds the concept of constant pointer & pointer to a constant pointer.

`char const *p2 = .HELLO.; // constant pointer`

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
Char	Character or small integer	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
Float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2bytes	1 wide character

2.3 VARIABLES IN C++

Declaration of variables.

C requires all the variables to be defined at the beginning of a scope. But c++ allows the declaration of variable anywhere in the scope. That means a variable can be declared right at the place of its first use. It makes the program easier to understand.

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier.

For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once

declared, the variables `a` and `mynumber` can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas.

For example: `int a, b, c;`

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as: `int a;`

`int b;`
 `int c;`

The integer data types `char`, `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name.

For example: `unsigned short int NumberOfSisters;`
 `signed int MyAccountBalance;`

By default, if we do not specify either `signed` or `unsigned` most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

`int MyAccountBalance;`

with exactly the same meaning (with or without the keyword `signed`)

An exception to this general rule is the `char` type, which exists by itself and is considered a different fundamental data type from signed `char` and unsigned `char`, thought to store characters. You should use either `signed` or `unsigned` if you intend to store numerical values in a `char`-sized variable.

`short` and `long` can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: `short` is equivalent to `short int` and `long` is equivalent to `long int`. The following two variable declarations are equivalent:

`short Year;`
 `short int Year;`

Finally, `signed` and `unsigned` may also be used as standalone type specifiers, meaning the same as `signed int` and

unsigned int respectively. The following two declarations are equivalent:

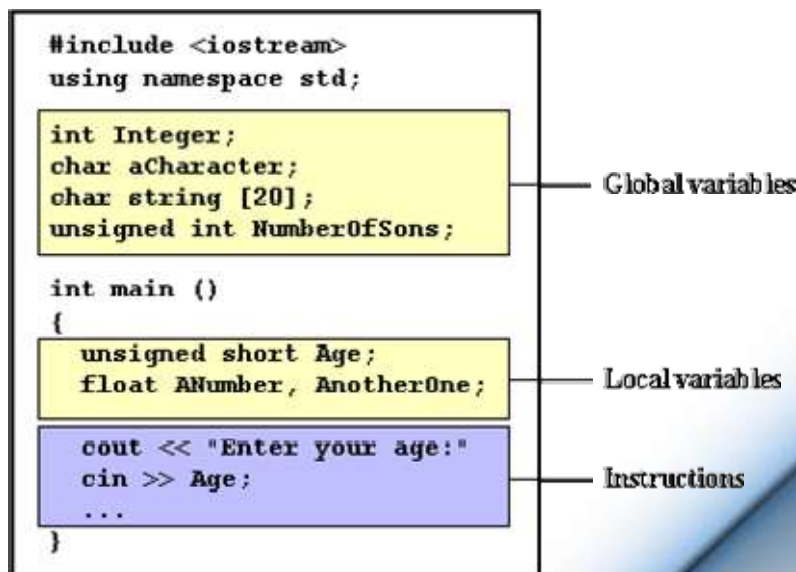
```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

Scope of variables:

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}), where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Dynamic initialization of variables.

In c++, a variable can be initialized at run time using expressions at the place of declaration. This is referred to as dynamic initialization.

Ex `int m = 10;`

Here variable m is declared and initialized at the same time.

Reference variables

i) A reference variable provides an alias for a previously defined variable.

ii) Example:-

If we make the variable sum a reference to the variable total, then sum & total can be used interchangeably to represent that variable.

iii) Reference variable is created as:-

data type & reference name = variable name.

Ex `int sum = 200;`
 `int &total = sum;`

Here total is the alternative name declared to represent the variable sum. Both variable refer to the same data object in memory.

iv) A reference variable must be initialized at the time of declaration.

v) C++ assigns additional meaning to the symbol &..Here & is not an address operation. The notation `int &` means reference to int. A major application of reference variable is in passing arguments to functions.

Ex `void fun (int &x) // uses reference`
 `{`
 `x = x + 10; // x increment, so x also incremented.`
 `}`
 `int main()`
 `{`
 `int n = 10;`
 `fun(n); // function call`
 `}`

When the function call `fun(n)` is executed, it will assign `x` to `n`. i.e. `int &x = n;`

Therefore `x` and `n` are aliases & when function increments `x`, `n` is also incremented. This type of function call is called call by reference.

Introduction to strings:

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace` statement). The following initialization formats are valid for strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution.

2.4 OPERATORS IN C++

2.4.1 Assignment (=):

The assignment operator assigns a value to a variable.

```
a=5;
```

This statement assigns the integer value 5 to the variable `a`. The part at the left of the assignment operator (`=`) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The *lvalue* has to be a variable whereas the *rvalue* can be a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a= b
```

This statement assigns to variable `a` (the *lvalue*) the value contained in variable `b` (the *rvalue*). The value that was stored until

this moment in `a` is not considered at all in this operation, and in fact that value is lost.

2.4.2 Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see may be *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3
```

The variable `a` will contain the value 2, since 2 is the remainder from dividing 11 between 3.

2.4.3 Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

2.4.4 Increase and decrease (++ , --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```


are all equivalent in its functionality: the three of them increase by one the value of c.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example1
B=3;
A=++B;

Example 2
B=3;
A=B++;

Output //A contains 4 //A contains 3, B contains 4

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

2.4.5 Relational and equality operators (==, !=, >, <, >=, <=):

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

Operators	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

```
(7 == 5)    // evaluates to false.
(5 > 4)     // evaluates to true.
(3 != 2)    // evaluates to true.
(6 >= 6)    // evaluates to true.
(5 < 5)     // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that $a=2$, $b=3$ and $c=6$,

```
(a == 5)    // evaluates to false since a is not equal to 5.
(a*b >= c)  // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.
```

Be careful! The operator $=$ (one equal sign) is not the same as the operator $==$ (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one ($==$) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression $((b=2) == a)$, we first assigned the value 2 to b and then we compared it to a , that also stores the value 2, so the result of the operation is true.

2.4.6 Logical operators (!, &&, ||):

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand.

For example:

```
!(5 == 5)    // evaluates to false because the expression at its right
(5 == 5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true       // evaluates to false
!false      // evaluates to true.
```

The logical operators $\&\&$ and $\|\|$ are used when evaluating two expressions to obtain a single relational result. The operator $\&\&$ corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator $\&\&$ evaluating the expression $a \&\& b$:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

|| OPERATOR

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

For example:
 ((5 == 5) && (3 > 6)) // evaluates to false (true && false).
 ((5 == 5) || (3 > 6)) // evaluates to true (true || false).

2.4.7 Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:
 condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.

7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
 7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
 5>3 ? a : b // returns the value of a, since 5 is greater than 3.
 a>b ? a : b // returns whichever is greater, a or b.

2.4.8 Comma operator (,):

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered. For example, the following code:
 a = (b=3, b+2);

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

2.4.9 sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a, because char is a one-byte long type.

The value returned by sizeof is a constant, so it is always determined before program execution.

Some of the new operators in c++ are-

- | | | |
|----|--------|--------------------------------|
| 1. | :: | Scope resolution operators. |
| 2. | ::* | pointer to member decelerator. |
| 3. | → * | pointer to member operator. |
| 4. | .* | pointer to member operator. |
| 5. | delete | memory release operator. |
| 6. | endl | Line feed operator |
| 7. | new | Memory allocation operator. |
| 8. | stew | Field width operator. |

Scope resolution Operator.

1)C++ is a block - structured language. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.

2) Consider following program.

```
...
....
{ int x = 1;
===
}
=====
{ int x = 2;
} =====
```

The two declaration of x refer to two different memory locations containing different values. Blocks in c++ are often nested. Declaration in a inner block hides a declaration of the same variable in an outer block.

3)In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by using scope resolution operator (::), because this operator allows access to the global version of a variable.

4) Consider following program.

```
#include<iostream.h >
int m = 10;
int main ( )
{
    int m = 20;
    {
        int k = m;
        int m = 30;
        cout << .K = . <<k;
        cout << .m = .<<m;
        cout << :: m = . << :: m;
    }
    cout << .m. = .<< m;
    cout << ::m = <<::m;
    return 0;
}
```

The output of program is

```
k = 20
m = 30
:: m = 10
m = 20
:: m=10
```

In the above program m is declared at three places. And ::m will always refer to global m.

5) A major application of the scope resolution operator is in the classes to identify the class to which a member functions belongs.

Member dereferencing operators .

C++ permits us to define a class containing various types of data & functions as members. C++ also permits us to access the class members through pointers. C++ provides a set of three pointer. C++ provides a set of three pointers to member operators.

- 1) ::* - To access a pointer to a member of a class.
- 2) .* - To access a member using object name & a pointer to that member.
- 3) →* - To access a member using a pointer in the object & a pointer to the member.

Memory management operators.

- 1) We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. C++ supports

two unary operators **new** and **delete** that perform the task of allocating & freeing the memory.

- 2) An object can be created by using new and destroyed by using delete. A data object created inside a block with new, will remain existence until it is explicitly destroyed by using delete.
- 3) It takes following form.

variable = new data type

The new operator allocated sufficient memory to hold a data object of type data-type & returns the address of the object.

EX p = new int.

Where p is a pointer of type int. Here p must have already been declared as pointer of appropriate types.

- 4) New can be used to create a memory space for any data type including user defined type such all array, classes etc.

Ex int * p = new int [10]

Creates a memory space for an array of 10 integers.

- 5) When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form is delete variable. If we want to free a dynamically allocated array, we must use following form.

delete [size] variable.

The size specifies the no of elements in the array to be freed.

- 6) The new operator has following advantages over the function malloc() in c -.
 - i) It automatically computes the size of the data object. No need to use sizeof()
 - ii) It automatically returns the correct pointer type, so that there is no need to use a type cast.
 - iii) new and delete operators can be overloaded.
 - iv) It is possible to initialize the object while creating the memory space.

Manipulators:

Manipulators are operators that are used to format the data display. There are two important manipulators.

- 1) endl
- 2) setw

1) endl :-

This manipulator is used to insert a linefeed into an output. It has same effect as using `.\n.` for newline.

Ex `cout<< .a. << a << endl <<.n=. <<n;
 << endl<<.p=.<<p<<endl;`

The output is

```
a = 2568
n = 34
p = 275
```

2) Setw :-

With the `setw`, we can specify a common field width for all the numbers and force them to print with right alignment.

EX `cout<<setw (5) <<sum<<endl;`

The manipulator `setw(5)` specifies a field width of 5 for printing the value of variable `sum` the value is right justified.

		3	5	6
--	--	---	---	---

Type cast operator.

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Syntax . `type name (expression)`

Ex `avg = sum/float(i)`

Here a type name behaves as if it is a function for converting values to a designated type.

□□□□

CONTROL STATEMENTS

Unit Structure

1. Introduction :
2. if – else Statement
3. Nested ifs :
4. The switch Statement
5. The for Loop
6. The while Loop
7. The do-while Loop
8. Using break to Exit a Loop
9. Using the goto Statement

1. INTRODUCTION :

This module discusses the statements that control a program's flow of execution. There are three categories of : selection statements, which include the if and the switch; iteration statements, which include the for, while, and do-while loops; and jump statements, which include break, continue, return, and goto. Except for return, which is discussed later in this book, the remaining control statements, including the if and for statements to which you have already had a brief introduction, are examined here.

2. IF – ELSE STATEMENT

The complete form of the if statement is
if(expression) statement
else statement

where the targets of the if and else are single statements. The else clause is optional. The targets of both the if and else can also be blocks of statements. The general form of the if using blocks of statements is

```

if(expression)
{
    statement sequence
}
else
{
    statement sequence
}

```

If the conditional expression is true, the target of the if will be executed; otherwise, the target of the else, if it exists, will be executed. At no time will both be executed. The conditional expression controlling the if may be any type of valid C++ expression that produces a true or false result.

This program uses the 'if' statement to determine whether the user's guess matches the magic number. If it does, the message is printed on the screen. Taking the Magic Number program further, the next version uses the else to print a message when the wrong number is picked:

```

#include<iostream>
#include<cstdlib>
int main()
{
    int magic;// magic number
    int guess;// users guess
    magic = rand(); // get a random number
    cout << "Enter your guess"
    cin >> guess;
    if (guess == magic) cout << "Right";
    else cout<< " sorry , your are wrong";
    return 0;
}

```

3.3 NESTED IFS :

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. The main thing to remember about nested ifs in C++ is that an else statement always refers to the nearest if statement that is within the same block as the else and not already associated with an else.

The general form of the nested if using blocks of statements is

```

if(expression)
{
    statement sequence
    if(expression)
    {
        statement sequence
    }
    else
    {
        statement sequence
    }
}
else
{
    statement sequence
}

```

Here is an example

```

if(i)
{
    if(j) result =1;
    if(k) result =2;
    else result = 3;
}
else    result = 4;

```

The final else is not associated with if(j) (even though it is the closest if without an else), because it is not in the same block. Rather, the final else is associated with if(i). The inner else is associated with if(k) because that is the nearest if. You can use a nested if to add a further improvement to the Magic Number program. This addition provides the player with feedback about a wrong guess.

3.4 THE SWITCH STATEMENT

The second of C++'s selection statements is the switch. The switch provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested if statements can perform multiway tests, for many situations the switch is a more efficient approach. It works like this: the value of

an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general form of the switch statement is Switch(expression)

```
{
    case constant 1 :
        Statement sequence
        Break;
    case constant 2 :
        Statement sequence
        Break;
    case constant 3 :
        Statement sequence
        Break;
    .
    .
    .
    Default:
        Statement sequence
}
```

The switch expression must evaluate to either a character or an integer value. (Floatingpoint expressions, for example, are not allowed.) Frequently, the expression controlling the switch is simply a variable. The case constants must be integer or character literals.

The default statement sequence is performed if no matches are found. The default is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that case are executed until the break is encountered or, in a concluding case or default statement, until the end of the switch is reached.

There are four important things to know about the switch statement:

The switch differs from the if in that switch can test only for equality (that is, for matches between the switch expression and the case constants), whereas the if conditional expression can be of any type.

No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.

A switch statement is usually more efficient than nested ifs.

The statement sequences associated with each case are not blocks. However, the entire switch statement does define a block. The importance of this will become apparent as you learn more about C++.

The following program demonstrates the switch. It asks for a number between 1 and 3, inclusive. It then displays a proverb linked to that number. Any other number causes an error message to be displayed.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout<< "Enter a number from 1 to 3";
    cin >> num;
    switch(num)
    {
        case 1:
            cout << " A rolling stone gathers no moss.\n";
            break;

        case 2:
            cout<< " A bird in hand is worth two in the bush.\n";
            break;

        case 3:
            cout<< " A full and his money soon parted.\n";
            break;

        default :
            cout<< " you must enter 1 , 2 or 3.\n";
            break;
    }
    return 0;
}
```

Here is a sample run

```
Enter a number from 1 to 32
A bird in hand is worth two in the bush.
```

3.5 THE FOR LOOP

You have been using a simple form of the for loop since Module 1. You might be surprised at just how powerful and flexible the for loop is. Let's begin by reviewing the basics, starting with the most traditional forms of the for.

The general form of the for loop for repeating a single statement is

```
for(initialization; expression; increment) statement;
```

For repeating a block, the general form is

```
for(initialization; expression; increment)
{
    statement sequence
}
```

The initialization is usually an assignment statement that sets the initial value of the loop control variable, which acts as the counter that controls the loop. The expression is a conditional expression that determines whether the loop will repeat. The increment defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The for loop will continue to execute as long as the conditional expression tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the for block.

The following program uses a for loop to print the square roots of the numbers between 1 and 99. Notice that in this example, the loop control variable is called num.

```
#include <iostream>
#include <cmath>
int main()
{
    int num;
    double sq_root;

    for(num=1; num<100; num++)
    {
        sq_root=sqrt((double)num);
        cout<<num<<" "<<sq_root<<"\n";
    }
    return 0;
}
```

This program uses the standard function `sqrt()`. As explained in Module 2, the `sqrt()` function returns the square root of its argument. The argument must be of type `double`, and the function returns a value of type `double`. The header `<cmath>` is required.

3.6 THE WHILE LOOP

Another loop is the `while`. The general form of the `while` loop is `while(expression) statement`; where `statement` may be a single statement or a block of statements. The expression defines the condition that controls the loop, and it can be any valid expression. The statement is performed while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The following program illustrates this characteristic of the `while` loop. It displays a line of periods. The number of periods displayed is equal to the value entered by the user. The program does not allow lines longer than 80 characters. The test for a permissible number of periods is performed inside the loop's conditional expression, not outside of it.

```
#include<iostream>
using namespace std;

int main()
{
    int len;
    cout<<"Enter length(1 to 79)";
    cin>>len;
    while(len>0 && len<80)
    {
        cout<<'.';
        len--;
    }
    return 0;
}
```

If `len` is out of range, then the `while` loop will not execute even once. Otherwise, the loop executes until `len` reaches zero. There need not be any statements at all in the body of the `while` loop. Here is an example: `while(rand() != 100) ;` This loop iterates until the random number generated by `rand()` equals 100.

7. THE DO-WHILE LOOP

The last of C++'s loops is the do-while. Unlike the for and the while loops, in which the condition is tested at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-while loop will always execute at least once. The general form of the do-while loop is

```
do { statements; } while(condition);
```

Although the braces are not necessary when only one statement is present, they are often used to improve readability of the do-while construct, thus preventing confusion with the while. The do-while loop executes as long as the conditional expression is true.

The following program loops until the number 100 is entered:

```
#include<iostream>
using namespace std;

int main()
{
    int num;
    do
    {
        cout << "Enter a number (100 to stop):";
        cin >> num;
    }
    while(num!=100);
    return 0;
}
```

8. USING BREAK TO EXIT A LOOP

It is possible to force an immediate exit from a loop, bypassing the loop's conditional test, by using the break statement. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. Here is a simple example:

```
#include<iostream>
using namespace std;

int main()
{
    int t,
```



```

    for(t=0;t<100;t++)
    {
        if(t==10) break;
        cout<<t<< ' ';
    }
    return 0;
}

```

The output from the program is shown here: 0 1 2 3 4 5 6 7 8 9

As the output illustrates, this program prints only the numbers 0 through 9 on the screen before ending. It will not go to 100, because the break statement will cause it to terminate early.

Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using continue. The continue statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop.

For example, the following program prints the even numbers between 0 and 100:

```

#include<iostream>
using namespace std;

int main()
{
    int x;
    for(x=0;x<100;x++)
    {
        if(x%2) continue;
        cout<<x<< ' ';
    }
    return 0;
}

```

Only even numbers are printed, because an odd number will cause the continue statement to execute, resulting in early iteration of the loop, bypassing the cout statement. Remember that the % operator produces the remainder of an integer division. Thus, when x is odd, the remainder is 1, which is true. When x is even, the remainder is 0, which is false. In while and do-while loops, a continue statement will cause control to go directly to the

conditional expression and then continue the looping process. In the case of the for, the increment part of the loop is performed, then the conditional expression is executed, and then the loop continues.

3.9 USING THE GOTO STATEMENT

The goto is C++'s unconditional jump statement. Thus, when encountered, program flow jumps to the location specified by the goto.

There are times when the use of the goto can clarify program flow rather than confuse it. The goto requires a label for operation. A label is a valid C++ identifier followed by a colon. Furthermore, the label must be in the same function as the goto that uses it.

For example, a loop from 1 to 100 could be written using a goto and a label, as shown here:

```
x=1;
loop1:
x++;
if(x<100) goto loop1;
```

One good use for the goto is to exit from a deeply nested routine. For example, consider the following code fragment:

```
for(...){
    for(...){
        while(...){
            if(...)goto stop
            .
            .
            .
        }
    }
}

stop:
cout<<"Enter in program.\n";
```

Eliminating the goto would force a number of additional tests to be performed. A simple break statement would not work here, because it would only cause the program to exit from the innermost loop.



FUNCTIONS

Unit Structure

1. Introduction :
2. Function
3. Arguments passed by value and by reference.
4. Default values in parameters
5. Overloaded functions
6. inline functions
7. Recursivity

1. INTRODUCTION :

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

2. FUNCTION

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements
}
```

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
    int r;
    r=a+b;
    return (r);
}

int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
    return 0;
}
```


The result is **8**

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable `z` of type `int`. Right after that, we see a call to a function called `addition`. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)

z = addition ( 5 , 3 );
```



The parameters and arguments have a clear correspondence. Within the main function we called to `addition` passing two values: 5 and 3, that correspond to the `int a` and `int b` parameters declared for function `addition`.

At the point at which the function is called from within main, the control is lost by main and passed to function `addition`. The

value of both arguments passed in the call (5 and 3) are copied to the local variables `int a` and `int b` within the function.

Function `addition` declares another local variable (`int r`), and by means of the expression `r=a+b`, it assigns to `r` the result of `a` plus `b`. Because the actual parameters passed for `a` and `b` are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

finalizes function `addition`, and returns the control back to the function that called it in the first place (in this case, `main`). At this moment the program follows its regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

```
int addition (int a, int b)
    ↓ 8
z = addition ( 5 , 3 );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to the value returned by `addition (5, 3)`, that is 8. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (8).

The following line of code in `main` is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

4.3 ARGUMENTS PASSED BY VALUE AND BY REFERENCE.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves.


For example, suppose that we called our first function `addition` using the following code:


```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int addition (int a, int b)

z = addition ( 5 , 3 );
```



This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
    a*=2;
    b*=2;
    c*=2;
}

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << " , y=" << y << " , z=" << z;
    return 0;
}
```

The output is : x=2, y=6, z=14

The first thing that should call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

```
void duplicate (int& a,int& b,int& c)
           x   y   z
duplicate ( x , y , z );
```

To explain it in another way, we associate `a`, `b` and `c` with the arguments passed on the function call (`x`, `y` and `z`) and any change that we do on `a` within the function will affect the value of `x` outside it. Any change that we do on `b` will affect `y`, and the same with `c` and `z`.

That is why our program's output, that shows the values stored in `x`, `y` and `z` after the call to `duplicate`, shows the values of all the three variables of `main` doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of `x`, `y` and `z` without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
using namespace std;
```

```

void prevnext (int x, int& prev, int& next)
{
    prev = x-1;
    next = x+1;
}

int main ()
{
    int x=100, y, z;
    prevnext (x, y, z);
    cout << "Previous=" << y << ", Next=" << z;
    return 0;
}

```

The output is :

Previous=99, Next=101

4.4 DEFAULT VALUES IN PARAMETERS

When declaring a function we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

For example:

```

// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}

int main ()
{
    cout << divide (12);
}

```

```

        cout << endl;
        cout << divide (20,4);
        return 0;
    }

```

The output is :

6
5

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is 6 ($12/2$).

In the second call:

divide (20,4)

there are two parameters, so the default value for b (`int b=2`) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 ($20/4$).

4.5 OVERLOADED FUNCTIONS

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters.

For example:

```

// overloaded function
#include <iostream>
using namespace std;

int operate (int a, int b)
{
    return (a*b);
}

float operate (float a, float b)

```

```

    {
        return (a/b);
    }

int main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << operate (x,y);
    cout << "\n";
    cout << operate (n,m);
    cout << "\n";
    return 0;
}

```

The output is :

10
2.5

In this case we have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two `ints` as its arguments it calls to the function that has two `int` parameters in its prototype and if it is called with two `floats` it will call to the one which has two `float` parameters in its prototype.

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

4.6 INLINE FUNCTIONS

The `inline` specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code

generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

4.7 RECURSIVITY

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

more concretely, 5! (factorial of 5) would be:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}

int main ()
```

```
{  
    long number;  
    cout << "Please type a number: ";  
    cin >> number;  
    cout << number << "! = " << factorial (number);  
    return 0;  
}
```

The output is : Please type a number: 9
 9! = 362880

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime). This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.



OBJECT AND CLASSES

Unit Structure

1. Introduction to Classes
2. Class Definition
3. Classes and Objects
4. Access specifiers- Private Public and Protected members
5. Member Functions of a Class
6. Passing and Returning Objects
7. Creating and Using a Copy Constructor

1. INTRODUCTION TO CLASSES

Object-oriented programming (OOP) is a conceptual approach to design programs. It can be implemented in many languages, whether they directly support OOP concepts or not. The C language also can be used to implement many of the object-oriented principles. However, C++ supports the object-oriented features directly. All these features like Data abstraction, Data encapsulation, Information hiding etc have one thing in common – the vehicle that is used to implement them. The vehicle is “class.”

Class is a user defined data type just like structures, but with a difference. It also has three sections namely private, public and protected. Using these, access to member variables of a class can be strictly controlled.

2. CLASS DEFINITION

The following is the general format of defining a class template:

```
class class_name
{
    permission_label_1:
    member1;
    permission_label_2:
    member2;
    ...
} object_name;
```

example:-

```

class tag_name
{
    public : // Must
    type member_variable_name;
    :
    type member_function_name();
    :
    private: // Optional
    type member_variable_name;
    :
    type member_function_name();
    :
};

```

The keyword class is used to define a class template. The private and public sections of a class are given by the keywords 'private' and 'public' respectively. They determine the accessibility of the members. All the variables declared in the class, whether in the private or the public section, are the members of the class. Since the class scope is private by default, you can also omit the keyword private. In such cases you must declare the variables before public, as writing public overrides the private scope of the class.

e.g.

```

class tag_name
{
    type member_variable_name; // private
    :
    type member_function_name(); // private
    :
    public : // Must
    type member_variable_name;
    :
    type member_function_name();
    :
};

```

The variables and functions from the public section are accessible to any function of the program. However, a program can access the private members of a class only by using the public member functions of the class. This insulation of data members from direct access in a program is called information hiding.

e.g.

```
class player
{
    public :
    void getstats(void);
    void showstats(void);
    int no_player;
    private :
    char name[40];
    int age;
    int runs;
    int tests;
    float average;
    float calcaverage(void);
};
```

The above example models a cricket player. The variables in the private section – name, age, runs, highest, tests, and average – can be accessed only by member functions of the class `calcaverage()`, `getstats()` and `showstats()`. The functions in the public section - `getstats()` and `showstats()` can be called from the program directly , but function `calcaverage()` can be called only from the member functions of the class – `getstats()` and `showstats()`.

With information hiding one need not know how actually the data is represented or functions implemented. The program need not know about the changes in the private data and functions. The interface(public) functions take care of this. The OOP methodology is to hide the implementation specific details, thus reducing the complexities involved.

5.3 CLASSES AND OBJECTS

As seen earlier, a class is a vehicle to implement the OOP features in the C++ language. Once a class is declared, an object of that type can be defined. An object is said to be a specific instance of a class just like Maruti car is an instance of a vehicle or pigeon is the instance of a bird. Once a class has been defined several objects of that type can be declared. For instance, an object of the class defined above can be declared with the following statement:


```
player Sachin, Dravid, Mohan ;
[Or]
class player Sachin , Dravid, Mohan ;
```

where Sachin and Dravid are two objects of the class player. Both the objects have their own set of member variables. Once the object is declared, its public members can be accessed using the dot operator with the name of the object. We can also use the variable `no_player` in the public section with a dot operator in functions other than the functions declared in the public section of the class.

e.g.

```
Sachin.getstats();
Dravid.showstats();
Mohan.no_player = 10;
```

5.4 ACCESS SPECIFIS- PRIVATE PUBLIC AND PROTECTED MEMBERS

Class members can either be declared in public', 'protected' or in the 'private' sections of the class. But as one of the features of OOP is to prevent data from unrestricted access, the data members of the class are normally declared in the private section. The member functions that form the interface between the object and the program are declared in public section (otherwise these functions can not be called from the program). The member functions which may have been broken down further or those, which do not form a part of the interface, are declared in the private section of the class. By default all the members of the class are private. The third access specifier 'protected' that is not used in the above example, pertains to the member functions of some new class that will be inherited from the base class. As far as non-member functions are concerned, private and protected are one and the same.

Summary of Access specifiers:

- private members of a class are accessible only from other members of their same class or from their "friend" classes.
- protected members are accessible from members of their same class and friend classes, and also from members of their derived classes.
- Finally, public members are accessible from anywhere the class is visible.

5.5 MEMBER FUNCTIONS OF A CLASS

A member function of the class is same as an ordinary function. Its declaration in a class template must define its return value as well as the list of its arguments. You can declare or define the function in the class specifier itself, in which case it is just like a normal function. But since the functions within the class specifier is considered inline by the compiler we should not define large functions and functions with control structures, iterative statements etc should not be written inside the class specifier.

However, the definition of a member function differs from that of an ordinary function if written outside the class specifier. The header of a member function uses the scope operator (::) to specify the class to which it belongs. The syntax is:

```
return_type class_name :: function_name (parameter list)
{
:
:
}
```

e.g.

```
void player :: getstats (void)
{
:
:
}

void player :: showstats (void)
{
:
:
}
```

This notation indicates that the functions getstats () and showstats() belong to the class player.

EXAMPLE OF CLASS: Find the area of the rectangle

```
// class example
#include <iostream.h>
class CRectangle
{
    int x, y;
    public:
    void set_values (int,int);
```

```

        int area (void) {return (x*y);}
    };

    void CRectangle::set_values (int a, int b)
    {
        x = a;
        y = b;
    }

    int main ()
    {
        CRectangle rect, rectb;

        rect.set_values (3,4);
        rectb.set_values (5,6);

        cout << "rect area: " << rect.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
    }

```

The output is :

```

        rect area: 12
        rectb area: 30

```

~~5.6 PASSING AND RETURNING OBJECTS~~

Objects can be passed to a function and returned back just like normal variables. When an object is passed by content , the compiler creates another object as a formal variable in the called function and copies all the data members from the actual variable to it. Objects can also be passed by address, which will be discussed later.

e.g.

```

class check
{
    public :
        check add(check);

        void get()
        {

```

```

        cin >> a;
    }

    void put()
    {
        cout << a;
    }
private :
    int a;
};

void main()
{
    check c1,c2,c3;
    c1.get();
    c2.get();
    c3 = c1.add(c2);
    c3.put();
}

check check :: add ( check c2)
{
    check temp;
    temp.a = a + c2.a;
    return ( temp);
}

```

The above example creates three objects of class check. It adds the member variable of two classes, the invoking class c1 and the object that is passed to the function , c2 and returns the result to another object c3.

You can also notice that in the class add() the variable of the object c1 is just referred as 'a' where as the member of the object passed .i.e. c2 is referred as 'c2.a' . This is because the member function will be pointed by the pointer named this in the compiler where as what we pass should be accessed by the extraction operator '.'. we may pass more than one object and also normal variable. we can return an object or a normal variable from the function. We have made use of a temporary object in the function add() in order to facilitate return of the object.

5.7 CREATING AND USING A COPY CONSTRUCTOR

To begin, let's review why you might need to explicitly define a copy constructor. By default, when an object is passed to a function, a bitwise (that is, exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object uses a resource, such as an open file, then the copy will use the same resource as does the original object. Therefore, if the copy makes a change to that resource, it will be changed for the original object, too!

Furthermore, when the function terminates, the copy will be destroyed, thus causing its destructor to be called. This may cause the release of a resource that is still needed by the original object.

A similar situation occurs when an object is returned by a function. The compiler will generate a temporary object that holds a copy of the value returned by the function. (This is done automatically and is beyond your control.) This temporary object goes out of scope once the value is returned to the calling routine, causing the temporary object's destructor to be called. However, if the destructor destroys something needed by the calling code, trouble will follow.

At the core of these problems is the creation of a bitwise copy of the object. To prevent them, you need to define precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creating a copy constructor.

Before we explore the use of the copy constructor, it is important for you to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization, which can occur three ways:

When one object explicitly initializes another, such as in a declaration.

When a copy of an object is made to be passed to a function.

When a temporary object is generated (most commonly, as a return value).

The copy constructor applies only to initializations. The copy constructor does not apply to assignments.

The most common form of copy constructor is shown here:

```
classname (const classname &obj)
{
    // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object. For example, assuming a class called MyClass, and y as an object of type MyClass, then the following statements would invoke the MyClass copy constructor:

```
MyClass x = y; // y explicitly initializing x
func1(y); // y passed as a parameter
y = func2(); // y receiving a returned object
```

In the first two cases, a reference to y would be passed to the copy constructor. In the third, a reference to the object returned by func2() would be passed to the copy constructor. Thus, when an object is passed as a parameter, returned by a function, or used in an initialization, the copy constructor is called to duplicate the object.

Remember, the copy constructor is not called when one object is assigned to another. For example, the following sequence will not invoke the copy constructor:

```
MyClass x;
MyClass y; x = y; // copy constructor not used here.
```

Again, assignments are handled by the assignment operator, not the copy constructor. The following program demonstrates a copy constructor:

```
#include <iostream>
using namespace std;

class MyClass
{
    int val;
    int copynumber;

    public:
    //Normal constructor
    MyClass(int i)
    {
```

```

        val=i;
        copynumber = 0;
        cout<< "Inside normal constructor\n";
    }

//Copy constructor

Myclass(const Myclass &o)
{
    val=o.val;
    copynumber=o.copynumber+1;
    cout<< " Inside copy constructor \n";
}

~Myclass()
{
    if (copynumber==0)
        cout<< "Destructing original \n";
    else
        cout<< "Destructing Copy"<< copynumber <<
        "\n";
}

int getval()
{
    return val;
}

};

void display(Myclass ob)
{
    cout<< ob.getval()<< "\n";
}

int main()
{
    Myclass a(10);
    display a;
    return 0;
}

```

The output is:

```
Inside normal constructor  
Inside copy constructor  
10  
Destructing copy 1  
Destructing original
```

Here is what occurs when the program is run: When `a` is created inside `main()`, the value of its copy number is set to 0 by the normal constructor. Next, `a` is passed to `ob` of `display()`. When this occurs, the copy constructor is called, and a copy of `a` is created. In the process, the copy constructor increments the value of copy number. When `display()` returns, `ob` goes out of scope. This causes its destructor to be called. Finally, when `main()` returns, `a` goes out of scope.

You might want to try experimenting with the preceding program a bit. For example, create a function that returns a `My Class` object, and observe when the copy constructor is called.



STRUCTURES AND UNION

Unit Structure

1. Structures
2. Pointers to structures
3. Nesting structures
4. User Defined Data Types
5. Unions
6. Enumerations (enum)

6.1 STRUCTURES

A data structure is a set of diverse types of data that may have different lengths grouped together under a unique declaration. Its form is the following:

```
struct model_name
{
    type1 element1;
    type2 element2;
    type3 element3;
    .
    .
} object_name;
```

where model_name is a name for the model of the structure type and the optional parameter object_name is a valid identifier (or identifiers) for structure object instantiations. Within curly brackets { } they are the types and their sub-identifiers corresponding to the elements that compose the structure.

If the structure definition includes the parameter model_name (optional), that parameter becomes a valid type name equivalent to the structure. For example:

```

struct products
{
    char name [30];
    float price;
};
products apple;
products orange, melon;

```

We have first defined the structure model `products` with two fields: `name` and `price`, each of a different type. We have then used the name of the structure type (`products`) to declare three objects of that type: `apple`, `orange` and `melon`.

Once declared, `products` has become a new valid type name like the fundamental ones `int`, `char` or `short` and we are able to declare objects (variables) of that type.

The optional field `object_name` that can go at the end of the structure declaration serves to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `orange` and `melon` this way:

```

struct products {
    char name [30];
    float price;
} apple, orange, melon;

```

Moreover, in cases like the last one in which we took advantage of the declaration of the structure model to declare objects of it, the parameter `model_name` (in this case `products`) becomes optional. Although if `model_name` is not included it will not be possible to declare more objects of this same model later.

It is important to clearly differentiate between what is a structure model, and what is a structure object. Using the terms we used with variables, the model is the type, and the object is the variable. We can instantiate many objects (variables) from a single model (type).

Once we have declared our three objects of a determined structure model (`apple`, `orange` and `melon`) we can operate with the fields that form them. To do that we have to use a point (`.`) inserted between the object name and the field name. For example, we

could operate with any of these elements as if they were standard variables of their respective types:

```
apple.name
apple.price
orange.name
orange.price
melon.name
melon.price
```

each one being of its corresponding data type: apple.name, orange.name and melon.name are of type char[30], and apple.price, orange.price and melon.price are of type float.

We are going to leave apples, oranges and melons and go with an example about movies:

// example about structures

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

struct movies_t {
    char title [50];
    int year;
} mine, yours;

void printmovie (movies_t movie);
int main ()
{
    char buffer [50];
    strcpy (mine.title, "2001 A Space Odyssey");

    mine.year = 1968;
    cout << "Enter title: ";
    cin.getline (yours.title,50);
    cout << "Enter year: ";
    cin.getline (buffer,50);
    yours.year = atoi (buffer);
```

```

        cout << "My favourite movie is:\n ";
        printmovie (mine);
        cout << "And yours:\n ";
        printmovie (yours);
        return 0;
    }

    void printmovie (movies_t movie)
    {
        cout << movie.title;
        cout << " (" << movie.year << ")\n";
    }

```

Output :

```

Enter title: Alien
Enter year: 1979
My favourite movie is:
2001 A Space Odyssey (1968)
And yours:
Alien (1979)

```

The example shows how we can use the elements of a structure and the structure itself as normal variables. For example, `yours.year` is a valid variable of type `int`, and `mine.title` is a valid array of 50 chars.

Notice that `mine` and `yours` are also treated as valid variables of type `movies_t` when being passed to the function `printmovie()`. Therefore, one of the most important advantages of structures is that we can refer either to their elements individually or to the entire structure as a block.

Structures are a feature used very often to build data bases, specially if we consider the possibility of building arrays of them.

```

// array of structures
#include <iostream.h>
#include <stdlib.h>
#define N_MOVIES 5

```

```

struct movies_t
{
    char title [50];
    int year;
} films [N_MOVIES];

void printmovie (movies_t movie);
int main ()
{
    char buffer [50];
    int n;
    for (n=0; n<N_MOVIES; n++)
    {
        cout << "Enter title: ";
        cin.getline (films[n].title,50);
        cout << "Enter year: ";
        cin.getline (buffer,50);
        films[n].year = atoi (buffer);
    }
    cout << "\nYou have entered these
movies:\n";
    for (n=0; n<N_MOVIES; n++)
        printmovie (films[n]);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year << ")\n";
}

```

Output :

```

Enter title: Alien
Enter year: 1979
Enter title: Blade Runner

```

```

Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Rear Window
Enter year: 1954
Enter title: Taxi Driver
Enter year: 1975
You have entered these movies:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)

```

6.2 POINTERS TO STRUCTURES

Like any other type, structures can be pointed by pointers. The rules are the same as for any fundamental data type: The pointer must be declared as a pointer to the structure:

```

struct movies_t
{
    char title [50];
    int year;
};
movies_t amovie;
movies_t * pmovie;

```

Here amovie is an object of struct type movies_t and pmovie is a pointer to point to objects of struct type movies_t. So, the following, as with fundamental types, would also be valid:

```
pmovie = &amovie;
```

Ok, we will now go with another example, that will serve to introduce a new operator:

```

// pointers to structures

#include <iostream.h>

```

```

#include <stdlib.h>
struct movies_t
{
    char title [50];
    int year;
};

int main ()
{
    char buffer[50];
    movies_t amovie;
    movies_t * pmovie;
    pmovie = & amovie;

    cout << "Enter title: ";
    cin.getline (pmovie->title,50);
    cout << "Enter year: ";
    cin.getline (buffer,50);

    pmovie->year = atoi (buffer);
    cout << "\nYou have entered:\n";
    cout << pmovie->title;
    cout << " (" << pmovie->year << ")\n";
    return 0;
}

```

Output:

```

Enter title: Matrix
Enter year: 1999
You have entered:
Matrix (1999)

```

The previous code includes an important introduction: operator `->`. This is a reference operator that is used exclusively with pointers to structures and pointers to classes. It allows us not to have to use parenthesis on each reference to a structure member. In the example we used:


```

pmovie->title
that could be translated to:
(*pmovie).title

```

both expressions `pmovie->title` and `(*pmovie).title` are valid and mean that we are evaluating the element `title` of the structure pointed by `pmovie`. You must distinguish it clearly from:

```

*pmovie.title
that is equivalent to
*(pmovie.title)

```

and that would serve to evaluate the value pointed by element `title` of structure `movies`, that in this case (where `title` is not a pointer) it would not make much sense.

6.3 NESTING STRUCTURES

Structures can also be nested so that a valid element of a structure can also be another structure.

```

struct movies_t
{
    char title [50];
    int year;
}

struct friends_t
{
    char name [50];
    char email [50];
    movies_t favourite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;

```

Therefore, after the previous declaration we could use the following expressions:

```
charlie.name
maria.favourite_movie.title
charlie.favourite_movie.year
pfriends->favourite_movie.year
```

(where, by the way, the last two expressions are equivalent).

The concept of structures that has been discussed in this section is the same as used in C language, nevertheless, in C++, the structure concept has been extended up to the same functionality of a class with the peculiarity that all of its elements are considered public.

6.4 USER DEFINED DATA TYPES

We have already seen a data type that is defined by the user (programmer): the structures. But in addition to these there are other kinds of user defined data types:

Typedef

C++ allows us to define our own types based on other existing data types. In order to do that we shall use keyword typedef, whose form is:

```
typedef existing_type new_type_name ;
```

where existing_type is a C++ fundamental or any other defined type and new_type_name is the name that the new type we are going to define will receive.

For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field [50];
```

In this case we have defined four new data types: C, WORD, string_t and field as char, unsigned int, char* and char[50] respectively, that we could perfectly use later as valid types:

```

C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;

```

Typedef can be useful to define a type that is repeatedly used within a program and it is possible that we will need to change it in a later version, or if a type you want to use has too long a name and you want it to be shorter.

6.5 UNIONS

Unions allow a portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```

union model_name
{
    type1 element1;
    type2 element2;
    type3 element3;
    .
    .
} object_name;

```

All the elements of the union declaration occupy the same space of memory. Its size is the one of the greatest element of the declaration.

For example:

```

union mytypes_t
{
    char c;
    int i;
    float f;
} mytypes;

```

defines three elements:
mytypes.c mytypes.i
mytypes.f

each one of a different data type. Since all of them are referring to a same location in memory, the modification of one of the elements will affect the value of all of them.

One of the uses a union may have is to unite an elementary type with an array or structures of smaller elements. For example,

```
union mix_t
{
    long l;
    struct
    {
        short hi;
        short lo;
    } s;
    char c[4];
} mix;
```

defines three names that allow us to access the same group of 4 bytes: mix.l, mix.s and mix.c and which we can use according to how we want to access it, as long, short or char respectively.

Anonymous unions

In C++ we have the option that unions be anonymous. If we include a union in a structure without any object name (the one that goes after the curly brackets { }) the union will be anonymous and we will be able to access the elements directly by its name. For example, look at the difference between these two declarations:

Union

```
struct
{
    char title[50];
    char author[50];
    union
    {
        float dollars;
        int yens;
    };
} book;
```

anonymous union

```

struct
{
    char title[50];
    char author[50];
    union
    {
        float dollars;
        int yens;
    } price;
} book;

```

The only difference between the two pieces of code is that in the first one we gave a name to the union (price) and in the second we did not. The difference is when accessing members dollars and yens of an object.

In the first case it would be:

```

book.price.dollars
book.price.yens

```

whereas in the second it would be:

```

book.dollars
book.yens

```

Once again I remind you that because it is a union, the fields dollars and yens occupy the same space in the memory so they cannot be used to store two different values. That means that you can include a price in dollars or yens, but not both.

6.6 ENUMERATIONS (ENUM)

Enumerations serve to create data types to contain something different that is not limited to either numerical or character constants nor to the constants true and false. Its form is the following:

```

enum model_name
{
    value1,
    value2,
    value3,
    .
    .
} object_name;

```


For example, we could create a new type of variable called color to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow,
white};
```

Notice that we do not include any fundamental data type in the declaration. To say it another way, we have created a new data type without it being based on any existing one: the type color_t, whose possible values are the colors that we have enclosed within curly brackets {}. For example, once declared the colors_t enumeration in the following expressions will be valid:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

In fact our enumerated data type is compiled as an integer and its possible values are any type of integer constant specified. If it is not specified, the integer value equivalent to the first possible value is 0 and the following ones follow a +1 progression. Thus, in our data type colors_t that we defined before, black would be equivalent to 0, blue would be equivalent to 1, green to 2 and so on.

If we explicitly specify an integer value for some of the possible values of our enumerated type (for example the first one) the following values will be the increases of this, for example:

```
enum months_t
{
    january=1, february, march, april,
    may, june, july, august,
    september, october, november, December
} y2k;
```

In this case, variable y2k of the enumerated type months_t can contain any of the 12 possible values that go from January to December and that are equivalent to values between 1 and 12, not between 0 and 11 since we have made January equal to 1.



ARRAYS

Unit Structure

1. Arrays
2. One-dimensional arrays
3. Two-Dimensional Arrays
4. Arrays of Strings
5. Arrays of Objects
6. Introduction to strings class

1. ARRAYS

An array is a collection of variables of the same type that are referred to by a common name. Arrays may have from one to several dimensions, although the one-dimensional array is the most common. Arrays offer a convenient means of creating lists of related variables.

The array that you will probably use most often is the character array, because it is used to hold a character string. The C++ language does not define a built-in string data type. Instead, strings are implemented as arrays of characters. This approach to strings allows greater power and flexibility than are available in languages that use a distinct string type.

2. ONE-DIMENSIONAL ARRAYS

A one-dimensional array is a list of related variables. Such lists are common in programming.

For example, you might use a one-dimensional array to store the account numbers of the active users on a network. Another array might store the current batting averages for a baseball team. When computing the average of a list of values, you will often use an array to hold the values. Arrays are fundamental to modern programming.

The general form of a one-dimensional array declaration is

```
type name[size];
```

Here, type declares the base type of the array. The base type determines the data type of each element that makes up the array. The number of elements the array can hold is specified by size. For example, the following declares an integer array named sample that is ten elements long:

```
int sample[10];
```

An individual element within an array is accessed through an index. An index describes the position of an element within an array. In C++, all arrays have zero as the index of their first element. Because sample has ten elements, it has index values of 0 through 9. You access an array element by indexing the array using the number of the element. To index an array, specify the number of the element you want, surrounded by square brackets. Thus, the first element in sample is sample[0], and the last element is sample[9].

For example, the following program loads sample with the numbers 0 through 9:

```
#include <iostream>
using namespace std;

int main();
{
    int sample[10]; // this reserves 10 integer
elements
    int t;

    //Load the array
    for(t=0; t<10; ++t)
        cout<<
sample["<<t<<"]:<<sample[t]<<"\n";
        "this is

    return 0;
}
```

The output from this example is shown here:

```
This is sample[0]: 0
This is sample[1]: 1
This is sample[2]: 2
This is sample[3]: 3
This is sample[4]: 4
This is sample[5]: 5
This is sample[6]: 6
This is sample[7]: 7
This is sample[8]: 8
This is sample[9]: 9
```

In C++, all arrays consist of contiguous memory locations. (That is, all array elements reside next to each other in memory.) The lowest address corresponds to the first element, and the highest address corresponds to the last element.

Arrays are common in programming because they let you deal easily with sets of related variables. Here is an example.

The following program creates an array of ten elements and assigns each element a value. It then computes the average of those values and finds the minimum and the maximum value.

```
#include <iostream>
using namespace std;

int main()
{
    int i, avg, min_val, max_val;
    int nums[10];

    num[0]=10;
    num[1]=18;
    num[2]=75;
    num[3]=0;
    num[4]=1;
```

```
num[5]=56;
num[6]=100;
num[7]=12;
num[8]=-19;
num[9]=88;
```

```
//Compute the average
```

```
avg=0;
for(i=0;i<10;i++)
{
    avg+=nums[i];
}
avg /=10;
cout<< "Average is" <<avg<< '\n';
```

```
//find minimum and maximum values
```

```
min_val=max_val=num[0];
for(i=1;i<10;i++)
{
    if(nums[i]<min_val)
    {
        min_val=nums[i];
    }
    if(nums[i]>max_val)
    {
        max_val=nums[i];
    }
}
cout<< "Minimum value:"<< min_val<< '\n';
cout<< "Maximum value:"<<max_val<< '\n';

return 0;
}
```

The output from the program is shown here:

```
Average is 34
Minimum value: -19
Maximum value: 100
```

Notice how the program cycles through the elements in the `nums` array. Storing the values in an array makes this process easy. As the program illustrates, the loop control variable of a `for` loop is used as an index. Loops such as this are very common when working with arrays.

7.3 TWO-DIMENSIONAL ARRAYS

C++ allows multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays.

To declare a two-dimensional integer array `twoD` of size 10,20, you would write

```
int twoD[10][20];
```

Pay careful attention to the declaration. Unlike some other computer languages, which use commas to separate the array dimensions, C++ places each dimension in its own set of brackets. Similarly, to access an element, specify the indices within their own set of brackets. For example, for point 3,5 of array `twoD`, you would use `twoD[3][5]`.

In the next example, a two-dimensional array is loaded with the numbers 1 through 12.

```
#include<iostream>
using namespace std;

int main()
{
    int t,i, nums[3][4];

    for(t=0;t<3;++t)
    {
        for(i=0;i<4;++i)
        {
```



```

        nums[t][i]=(t*4)+i+1;
        cout<<nums[t][i]<< ' ';
    }
    cout<< '\n';
}
return 0;
}

```

In this example, `nums[0][0]` will have the value 1, `nums[0][1]` the value 2, `nums[0][2]` the value 3, and so on. The value of `nums[2][3]` will be 12. Conceptually, the array will look like that shown here:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

← `num[1][2]`

7.4 ARRAYS OF STRINGS

A special form of a two-dimensional array is an array of strings. It is not uncommon in programming to use an array of strings. The input processor to a database, for instance, may verify user commands against a string array of valid commands. To create an array of strings, a two-dimensional character array is used, with the size of the left index determining the number of strings and the size of the right index specifying the maximum length of each string, including the null terminator. For example, the following declares an array of 30 strings, each having a maximum length of 79 characters plus the null terminator.

```
char str_array[30][80];
```

Accessing an individual string is quite easy: you simply specify only the left index. For example, the following statement calls `gets()` with the third string in `str_array`:

```
gets(str_array[2]);
```

To access an individual character within the third string, you will use a statement like this:

```
cout << str_array[2][3];
```

This displays the fourth character of the third string.

The following program demonstrates a string array by implementing a very simple computerized telephone directory. The two-dimensional array `numbers` holds pairs of names and numbers. To find a number, you enter the name. The number is displayed.

```
#include<iostream>
#include<cstring>
using namespace std;

int main()
{
    int i;
    char str[80];
    char numbers[10][80]= {    "Tom", "555-3322",
                                "Mary", "555-8976",
                                "Jon", "555-1037",
                                "Rachel", "555-1400"
                                };

    cout<< "Enter name:";
    cin>> str;

    for(i=0;i<10;i+=2)
    {
        if(!strcmp(str,numbers[i]))
        {
            cout<< "Number is " <<numbers[i+1] << "\n";
            break;
        }
        if(i==10)
        {
            cout<< "Not found\n";
        }
    }
    return 0;
}
```

Here is a sample run:

```
Enter name: Jon
Number is 555-1037
```

Notice how the for loop increments its loop control variable, *i*, by 2 each time through the loop. This is necessary because names and numbers alternate in the array.

7.5 ARRAYS OF OBJECTS

You can create arrays of objects in the same way that you create arrays of any other data type. For example, the following program creates an array of `MyClass` objects. The objects that comprise the elements of the array are accessed using the normal array-indexing syntax.

```
#include<iostream>
using namespace std;

class MyClass
{
    int x;

    public:
    void set_x(int i) {x=i;}
    int get_x() {return x;}
};

int main()
{
    MyClass obs[4];
    int i;

    for(i=0;i<4;i++)
    {obs[i].set_x(i);}

    for(i=0;i<4;i++)
    {
        cout<<          "obs["          "<<i<<"].get_x():"
"<<obs[i].get_x()<< "\n";
    }
    return 0;
}
```

This program produces the following output:

```
obs[0].get_x(): 0
obs[1].get_x(): 1
obs[2].get_x(): 2
obs[3].get_x(): 3
```

7.6 INTRODUCTION TO STRINGS CLASS

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace std;` statement).

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string mystring = "This is a string";
    cout << mystring;
    return 0;
}
```

The output is :

This is a string

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

```
// my first string
#include <iostream>
#include <string>
using namespace std;
```

```
int main ()
{
    string mystring;
    mystring = "This is the initial string content";
    cout << mystring << endl;
    mystring = "This is a different string content";
    cout << mystring << endl;
    return 0;
}
```

The output is :

```
This is the initial string content
This is a different string content
```



OPERATOR OVERLOADING

Unit Structure

1. Operator Overloading
2. Defining operator overloading
3. Unary Operator over loading -
4. Binary Operator overloading

8.1 OPERATOR OVERLOADING

Overloading means assigning different meaning to an operation depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to the operators.

Example:-

The input/output operators << and >> are good examples of operator overloading although the built in definition of the << operator is for shifting of bits. It is also used for displaying the values of various data types.

We can overload all the C++ operators except the following,

1. Class member access operators (., .*)
2. Scope resolution operator (: :)
3. Size operator (sizeof)
4. Conditional operator (?:)

8.2 DEFINING OPERATOR OVERLOADING

i) To define an additional task to an operator, operator function is used.

The general form is .

```
Return type classname :: operator op (anglist)
{
    Function body //task defined
}
```


Where return type is the type of value returned by the specified operation and op is the operator being overloaded. Operator op is the function name.

- ii) Operator functions must be either member functions or friend function. The difference is a friend function will have only one argument for unary operators and two for binary operators. While a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not case with friend function, because arguments may be passed either by value or by reference.

iii) Example .

```
Vector operator . ( ) ; // unary minus
Vector operator + (vector); // vector addition
Friend vector operator + (vector, vector) // vector
addition
Friend vector operator . (vector); // unary minus
Vector operator . (vector & a); // subtraction
Int operator == (vector); // comparision
Friend int operator == (vector,vector) //comparision
```

- iv) The process of overloading involves .

- a) Create a class that defines the data type that is to be used in the overloading operation.
- b) Declare the operator function operator op () in the public part of the class. It may be either a member or friend function.
- c) Define the operator function to implement the required operation.

- v) Overloaded operator functions can be invoked by expressions such as.

Op x or x op

For unary operators and

x op y

for binary operators.

And op x would be interpreted as
operator op (x)

for friend functions and $x \text{ op } y$ would be interpreted as $x.\text{operator op}(y)$

in member function and
operator $\text{op}(x,y)$ in friend function.

8.3 UNARY OPERATOR OVER LOADING -

- i) In overloading unary operator, a friend function will have only one argument, while a member function will have no arguments.
- ii) Let us consider the unary minus operator. A minus operator, when used as a unary takes just one operand.
- iii) This operator changes the sign of an operand when applied to a basic data item.

Following example shows how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variables.

```
# include <iostream.h>
class unary
{
    int x, y, z;
public:
    void getdata (int a, int b, int c);
    void display (void);
    void operator . (); // overload unary minus.
};
void unary :: getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void unary :: display (void)
{
    cout << x << " " << y << " " << z << "\n.";
}
void unary ::operator .()
{

```

```

x = -x ;
y = -y ;
z = -z ;
}

main ( )
{
    unary u;
    u.getdata(20, -30, 40);
    cout << . u : . ;
    u. display ( ) ;
    -u;
    cout << . u : . ;
    u. display ( ) ;
}

```

The output is : -

```
u : 20 -30 40
u : -20 30 -40
```

- iv) The function operator . () takes no argument because this function is a member function of the same class, it can directly access the member of the object which activated it.
- v) It is possible to overload a unary minus operator using a friend function as follows :

```

Friend void operator . (unary & u);
Void operator . (unary & u)
{
    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
}

```

The argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator(). Therefore changes made inside the operator function will not reflect in the called object.

8.4 BINARY OPERATOR OVERLOADING

- i) In overloading binary operator, a friend function will have two arguments, while a member function will have one argument.
- ii) Following example shows how to overload + operator to add 2 complex number

```
#include<iostream.h>
class complex
{
    float x, y;
    public :
    complex ( );
    complex (float real, float imag)
{
    x = real;
    y = imag;
}

    complex operator +(complex);
    void display(void);
};

    complex complex:: operator+(complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
    return (temp);
}
void complex : : display (void)
{
    cout << x << " + j " << y << " \n ";
}
main ( )
{
    complex c1,c2,c3;
    c1 = complex (2.5, 3.5);
    c2 = complex (1.6, 2.7);
    c3 = c1 + c2;
    cout << "c1 = "; c1.display( ) ;
    cout << "c2 = "; c2.display( ) ;
    cout << "c3 = "; c3.display( ) ;
}
```

The output is

```
c1 = 2.5 + j 3.5
c2 = 1.6 + j 2.7
c3 = 4.1 + j 6.2
```

- iii) In the above program, the function operator +, is expected to add two complex values and return a complex value as a result but receives only one value as argument.
- iv) The function is executed by the statement `c3 = c1 + c2`
Here, the object `c1` is used to invoke the function and `c2` plays the role of an argument that is passed to the function. The above statement is equivalent to `c3 = c1.operator +(c2)`

Here, in the operator `+()` function, the data members of `c1` are accessed directly and the data member of `c2` are accessed using the dot operator. Thus in overloading binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

Type Casting

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of `a` has been promoted from `short` to `int` and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int...`), to or from `bool`, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions.

For example:

```
class A {};
class B { public: B (A a) {} };
A a;
B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.

For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy
{
    float i,j;
};

class CAddition
{
    int x,y;
public:
    CAddition (int a, int b)
    {
        x=a;
        y=b;
    }
}
```



```

int result()
{
    return x+y;}

};

int main ()
{
    CDummy d;
    CAddition * padd;
    padd = (CAddition*) &d;
    cout << padd->result();
    return 0;
}

```

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result.

Pitfalls in automatic type conversion

Because the compiler must choose how to quietly perform a type conversion, it can get into trouble if you don't design your conversions correctly. A simple and obvious situation occurs with a class **X** that can convert itself to an object of class **Y** with an **operator Y()**. If class **Y** has a constructor that takes a single argument of type **X**, this represents the identical type conversion. The compiler now has two ways to go from **X** to **Y**, so it will generate an ambiguity error when that conversion occurs:

// Ambiguity in type conversion

```
class Y; // Class declaration
```

```
class X
```

```
{
    public:
    operator Y() const; // Convert X to Y
};
```

```
class Y
```

```
{
    public:
    Y(X); // Convert X to Y
}
```

```
};

void f(Y);

int main()
{
    X x;
    //! f(x); // Error: ambiguous conversion
} ///:~
```

The obvious solution to this problem is not to do it: Just provide a single path for automatic conversion from one type to another.

A more difficult problem to spot occurs when you provide automatic conversion to more than one type. This is sometimes called *fan-out*.

// Type conversion fanout

```
class A {};
class B {};

class C
{
    public:
        operator A() const;
        operator B() const;
};

// Overloaded h():
void h(A);
void h(B);

int main()
{
    C c;
    //! h(c); // Error: C -> A or C -> B ???
} ///:~
```

Class **C** has automatic conversions to both **A** and **B**. The insidious thing about this is that there's no problem until someone innocently comes along and creates two overloaded versions of **h()**. (With only one version, the code in **main()** works fine.)

Again, the solution – and the general watchword with automatic type conversion – is to only provide a single automatic conversion from one type to another. You can have conversions to

other types; they just shouldn't be *automatic*. You can create explicit function calls with names like **make_A()** and **make_B()**.

What is *explicit* keyword?

- A constructor that takes a single argument operates as an implicit conversion operator by default. This is also referred as converting constructor.
- To prevent this implicit conversion keyword *explicit* has been introduced in C++. This makes the constructor as non-converting.
- This keyword has effect only when defined on a single argument constructor or on a constructor which has default arguments for all but one argument.
- Compile time error will be reported if an attempt is made to create an object via conversion as in statements like "MyClass obj2 = 200;".

EXAMPLE: Demonstrate the usage of *explicit* keyword

```
#include <iostream>
using namespace std;

class MyClass
{
    int data;
public:
    explicit MyClass(int aData) {
        cout << "Constructor invoked" << endl;
        data = aData;
    }
};

int main ()
{
    MyClass obj1(100);

    /*
    * Object creation by conversion as below reports a compiler error.
    * Error E2034 explicit.cpp 15: Cannot convert 'int' to 'MyClass'
    * in function main()
    */
    // MyClass obj2 = 200;
}
```



INHERITANCE

Unit Structure

1. Introduction
2. Types of Inheritance
3. Derived and Base Class
4. Public Inheritance
5. Private Inheritance

9.1 INTRODUCTION :

C++ supports the concept of reusability once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called ***inheritance***.

The old class is referred to as base class or super class. And the new one is called as derived class or subclass.

Inheritance:

The Inheritance is the mechanism of deriving one class from the other class. Let us assume that the class def is derived from the existing base class abc. In that case abc is called as base class and def is called as derived class. The feature that is facility of the base class will be inherited to the derived class. The derived class will have its own features. The features of the derived class will never be inherited to the base class.

Need of Inheritance:

In object oriented programming there are applications for which the classes are develop, every classes will have its own features .Consider a situation where we require all the features of a particular existing class and we also need some additional features. The above requirement can be satisfied with three different methods.

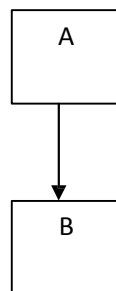
1) Modify the original existing class by acting new features into it. The disadvantage of this method is that the users of the original class will get some extra facility and they can misuse them

2) Rewrite the new class with all the required features i.e. the features of the existing class plus the new features. The disadvantage of this method is that it will be consuming more time and space.

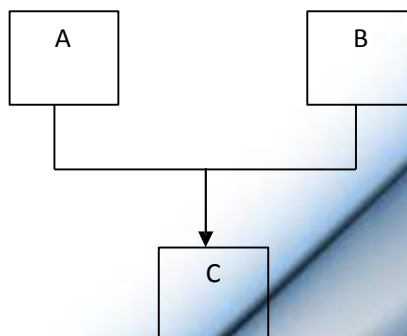
3) Derive the new class from the existing class and only write the new features in the derived class. The features of the original class will be inherited to the new class. This method consumes less space and time. Moreover the original class is not modified. Hence user of original class will not get extra facilities.

9.2 TYPES OF INHERITANCE: -

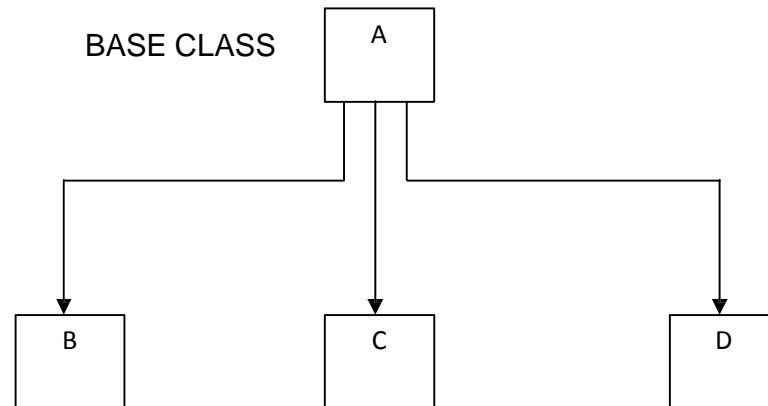
1. **Single Inheritance** : A derived class with only one base class, is called Single Inheritance.



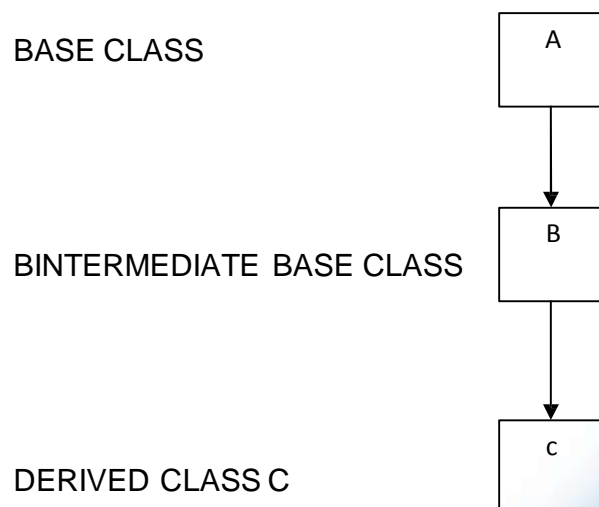
2. **Multiple Inheritance** : A derived class with several base classes, is called multiple Inheritance.



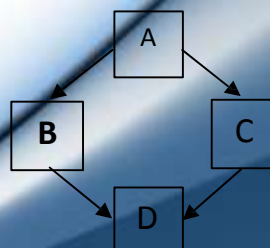
3. **Hierarchical Inheritance** : The properties of one class may be inherited by more than one class is called hierarchical Inheritance.



4. **Multilevel Inheritance** : The mechanism of deriving a class from another derived class is known as multilevel Inheritance.



5. **Hybrid Inheritance** : The hybrid Inheritance is a combination of all types of inheritance.



9.3 DERIVED AND BASE CLASS

1. Base class: -

A base class can be defined as a normal C++ class, which may consist of some data and functions.

Ex .

```
class Base
{
    :::::
}
```

2. Derived Class: -

- i) A class which acquires properties from base class is called derived class.
- ii) A derived class can be defined by class in addition to its own detail.

The general form is .

```
class derived-class-name: visibility-mode
base-class-name
{
    :::::
}
```

The colon indicates that the derived class name is derived from the base-classname.

The visibility-mode is optional and if present it is either private or public.

9.4 PUBLIC INHERITANCE

- i) When the visibility-mode is public the base class is publicly inherited.
- ii) In public inheritance, the public members of the base class become public members of the derived class and therefore they are accessible to the objects of the derived class.
- iii) Syntax is .

```
class ABC : public PQR
{
    members of ABC
};
```

Where PQR is a base class illustrate public inheritance.

iv) Let us consider a simple example to illustrate public inheritance.

```
#include<iostream.h>
class base
{
    int no1;
    public:
    int no2;
    void getdata();
    int getno1();
    void showno1();
};
class derived: public Base // public derivation.
{
    int no3;
    public:
    void add();
    void display();
};
void Base :: getdata()
{
    no1 = 10;
    no2 = 20;
}
int Base :: getno1()
{
    return no1;
}
void Base :: showno1()
{
    cout <<.Number1 =.<<no1 <<.\n.;
}
void Derived :: add()
{
    no3 = no2 + getno1(); // no1 is private
}
void Derived :: display()
{
    cout << .Number1 = .<<getno1() <<.\n.;
    cout << .Number2 = .<<no2<<.\n.;
```

```

        cout << .Sum . <<no3 << .\n.;
    }
    main ( )
    {
        derived d;
        d.getdata ( );
        d.add ( );
        d.showno1 ( );
        d.display ( ) ;
        d.b = 100;
        d.add ( ) ;
        d.display ( ) ;
        return 0;
    }

```

The output of the program is

```

Number1 = 10
Number1 = 10
Number2 = 20
Sum = 30
Number1 = 10
Number2 = 100
Sum = 110

```

- v) In the above program class Derived is public derivation of the base class Base.

Thus a public member of the base class Base is also public member of the derived class Derived.

9.5 PRIVATE INHERITANCE

- i) When the visibility mode is private, the base class is privately inherited.
- ii) When a base class is privately inherited by a derived class, public members of the base class becomes private members of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

iii) Public member of a class can be accessed by its own objects using the dot operator. So in private Inheritance, no member of the base class is accessible to the objects of the derived class.

iv) Syntax is .

```
class ABC: private PQR
{
    member of ABC
}
```

v) Let us consider a simple example to illustrate private inheritance.

```
#include <iostream.h>
class base
{
    int x;
    public :
    int y;
    void getxy() ;
    int get_x(void) ;
    void show_x(void) ;
};
void base ::getxy(void)
{
    cout<<".Enter Values for x and y : . ;
    cin >> x >> y ;
}
int base : : get_x()
{
    return x;
}
void base :: show_x()
{
    cout <<".x = . << x << .\n.;
}
void Derived : : mul( )
{
    getxy();
    z = y * get_x ( ) ;
}
void Derived ::display()
```

```

{
    show_x ( ) ;
    cout <<.y = . <<y<<.\n.;
    cout<<.Z = . <<z<<.\n.;
}
main ( )
{
    Derived d;
    d. mul();
    d.display() ;
    //d.y = 4a; won.t work y has become
    private.
    d.mul();
    d.display();
}

```

The output is Output: -

Enter values for x and y: 510

X = 5

Y = 10

Z = 50

Enter values for x and y:1220

X = 12

Y = 20

Z = 240

Derived Class Constructor

When a base class has a constructor, the derived class must explicitly call it to initialize the base class portion of the object. A derived class can call a constructor defined by its base class by using an expanded form of the derived class' constructor declaration.

The general form of this expanded declaration is shown here:

```

derived-constructor(arg-list) : base-cons(arg-list);
{
    body of derived constructor
}

```

Here, base-cons is the name of the base class inherited by the derived class. Notice that a colon separates the constructor declaration of the derived class from the base class constructor. (If a class inherits more than one base class, then the base class constructors are separated from each other by commas.)

The following program shows how to pass arguments to a base class constructor. It defines a constructor for TwoDShape that initializes the width and height properties.

```
// Add a constructor to TwoDshape.

#include<iostream>
#include<cstring>
using namespace std;

// A class for two-dimensional objects.
class TwoDshape
{
    //These are private
    double width;
    double height;
public:
    //Constructor for TwoDshape.
    TwoDshape(double w, double h)
    {
        width=w;
        height=h;
    }
    void showdim()
    {
        cout<<"Width and height are"<< width<<
        "and" << height<<"\n";
    }

    //accessor functions
    double getWidth() {return width;}
    double getHeight(){return height;}
    void setWidth(double w){width=w;}
    void setHeight(double h){height=h;}
};
```



```

//Triangle is derived from TwoDshape.
class Triangle : public TwoDshape
{
    char style[20]; // now private
public:
    // Constructor for triangle.
    Triangle(char *str, double w, double h) :
TwoDshape(w, h)
    {
        strcpy(style, str);
    }

    double area()
    {
        return getWidth() *getHeight()/2;
    }

    void showStyle()
    {
        cout<<"Triangle is"<<style<<"\n";
    }
};

int main()
{
    Triangle t1("isosceles",4.0,4.0);
    Triangle t2("right",8.0, 12.0);
    cout<<"Info for t1:\n";
    t1.showStyle();
    t1.showDim();
    cout<<"Area is" << t1.area()<<"\n";

    cout<<"\n";
    cout<<"Info for t2:\n";
    t2.shwoStyle();
    t2.showDim();
    cout<<"Area is"<<t2.area()<<"\n";

    return 0;
}

```

Here, `Triangle()` calls `TwoDShape` with the parameters `w` and `h`, which initializes width and height using these values. `Triangle` no longer initializes these values itself. It need only initialize the value unique to it: `style`. This leaves `TwoDShape` free to construct its subobject in any manner that it so chooses. Furthermore, `TwoDShape` can add functionality about which existing derived classes have no knowledge, thus preventing existing code from breaking.

Any form of constructor defined by the base class can be called by the derived class' constructor. The constructor executed will be the one that matches the arguments

Summary of access privileges

1. If the designer of the base class wants no one, not even a derived class to access a member, then that member should be made private.
2. If the designer wants any derived class function to have access to it, then that member must be protected.
3. if the designer wants to let everyone, including the instances, have access to that member , then that member should be made public.

Summary of derivations

- 1.Regardless of the type of derivation, private members are inherited by the derived class, but cannot be accessed by the new member function of the derived class , and certainly not by the instances of the derived class .
- 2.In a private derivation, the derived class inherits public and protected members as private. a new members function can access these members, but instances of the derived class may not. Also any members of subsequently derived classes may not gain access to these members because of the first rule.
- 3.In public derivation, the derived class inherits public members as public , and protected as protected . a new member function of the derived class may access the public and protected members of the base class ,but instances of the derived class may access only the public members.
- 4.In a protected derivation, the derived class inherits public and protected members as protected .a new members function of the derived class may access the public and protected members of the base class, both instances of the derived class may access only the public members .

Table of Derivation and access specifiers

Derivation Type	Base Member	Class	Access in Derived Class
Private	Private		(inaccessible)
	Public		Private
	Protected		Private
Public	Private		(inaccessible)
	Public		Public
	Protected		Protected
Protected	Private		(inaccessible)
	Public		Protected
	Protected		Protected

We can summarize the different access types according to whom can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
Not-members	yes	no	no

where "not-members" represent any reference from outside the class, such as from main(), from another class or from any function, either global or local.



POINTERS

Unit Structure

1. Introduction :
2. Pointer
3. The Pointer Operators
4. The Base Type of a Pointer
5. Assigning Values through a Pointer
6. Pointers and Arrays
7. Arrays of Pointers
8. Function Pointer
9. Assign an Address to a Function Pointer
10. Calling a Function using a Function Pointer
11. Pass a Function Pointer as an Argument
12. Return a Function Pointer
13. Pointers to Objects
14. Operators new and new[]
15. Operator delete and delete[]

10.1 INTRODUCTION ·

The pointer is one of C++'s most powerful features. It is also one of its most troublesome. Despite their potential for misuse, pointers are a crucial part of C++ programming. For example, they allow C++ to support such things as linked lists and dynamic memory allocation. They also provide one means by which a function can alter the contents of an argument. However, these and other uses of pointers will be discussed in subsequent modules. In this module, you will learn the basics about pointers and see how to manipulate them.

In a few places in the following discussions, it is necessary to refer to the size of several of C++'s basic data types. For the sake of discussion, assume that characters are one byte in length, integers are four bytes long, floats are four bytes long, and doubles have a length of eight bytes. Thus, we will be assuming a typical 32-bit environment.

2. POINTER

A pointer is an object that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to “point to” y.

Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer’s base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable. For example, to declare ip to be a pointer to an int, use this declaration:

int *ip;

Since the base type of ip is int, it can be used to point to int values. Here, a float pointer is declared:

float *fp;

In this case, the base type of fp is float, which means that it can be used to point to a float value. In general, in a declaration statement, preceding a variable name with an * causes that variable to become a pointer.

3. THE POINTER OPERATORS

There are two special operators that are used with pointers: * and &. The & is a unary operator that returns the memory address of its operand. (Recall that a unary operator requires only one operand.)

For example,

ptr = &total;

This puts into ptr the memory address of the variable total. This address is the location of total in the computer’s internal memory. It has nothing to do with the value of total. The operation of & can be remembered as returning “the address of” the variable it precedes. Therefore, the preceding assignment statement could be verbalized as “ptr receives the address of total.” To better understand this assignment, assume that the variable total is located at address 100. Then, after the assignment takes place, ptr has the value 100.

The second operator is *, and it is the complement of &. It is a unary operator that returns the value of the variable located at the

address specified by its operand. Continuing with the same example, if ptr contains the memory address of the variable total, then

```
val = *ptr;
```

will place the value of total into val. For example, if total originally had the value 3,200, then val will have the value 3,200, because that is the value stored at location 100, the memory address that was assigned to ptr. The operation of * can be remembered as “at address.” In this case, then, the statement could be read as “val receives the value at address ptr.”

The following program executes the sequence of the operations just described:

```
#include <iostream>
using namespace std;

int main()
{
    int total;
    int *ptr;
    int val;
    total = 3200; // assign 3,200 to total
    ptr = &total; // get address of total
    val = *ptr; // get value at that address
    cout << "Total is: " << val << '\n';
    return 0;
}
```

It is unfortunate that the multiplication symbol and the “at address” symbol are the same. This fact sometimes confuses newcomers to the C++ language. These operators have no relationship to each other. Keep in mind that both & and * have a higher precedence than any of the arithmetic operators except the unary minus, with which they have equal precedence. The act of using a pointer is often called indirection because you are accessing one variable indirectly through another variable.

10.4 THE BASE TYPE OF A POINTER

In the preceding discussion, you saw that it was possible to assign val the value of total indirectly through a pointer. At this point, you may have thought of this important question: How does C++ know how many bytes to copy into val from the address pointed to by ptr? Or, more generally, how does the compiler transfer the proper number of bytes for any assignment involving a pointer? The answer is that the base type of the pointer determines

the type of data upon which the pointer operates. In this case, because `ptr` is an `int` pointer, four bytes of information are copied into `val` (assuming a 32-bit `int`) from the address pointed to by `ptr`. However, if `ptr` had been a double pointer, for example, then eight bytes would have been copied.

It is important to ensure that pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type `int`, the compiler assumes that anything it points to will be an integer variable. If it doesn't point to an integer variable, then trouble is usually not far behind!

For example, the following fragment is incorrect:

```
int *p; double f; // ... p = &f; // ERROR
```

This fragment is invalid because you cannot assign a double pointer to an integer pointer. That is, `&f` generates a pointer to a double, but `p` is a pointer to an `int`. These two types are not compatible. (In fact, the compiler would flag an error at this point and not compile your program.)

Although two pointers must have compatible types in order for one to be assigned to another, you can override this restriction (at your own risk) using a cast.

For example, the following fragment is now technically correct: **26**
C++ A Beginner's Guide by Herbert Schildt

```
int *p ; double f; // ... p = (int *) &f; // Now
technically OK
```

The cast to `int *` causes the double pointer to be converted to an integer pointer. However, to use a cast for this purpose is questionable practice. The reason is that the base type of a pointer determines how the compiler treats the data it points to. In this case, even though `p` is actually pointing to a floating-point value, the compiler still "thinks" that `p` is pointing to an `int` (because `p` is an `int` pointer).

To better understand why using a cast to assign one type of pointer to another is not usually a good idea, consider the following short program:

```
#include<iostream>
using namespace std;

int main()
{
    double x, y;
```

```

        int *p;

        x=123.23;
        p=(int *);
        y=*p;          // This statement wont yield the
desire result

        cout<<y;
        return 0;
    }

```

Here is the output produced by the program. (You might see a different value.)

1.37439e+009

This value is clearly not 123.23! Here is why. In the program, p (which is an integer pointer) has been assigned the address of x (which is a double). Thus, when y is assigned the value pointed to by p, y receives only four bytes of data (and not the eight required for a double value), because p is an integer pointer. Therefore, the cout statement displays not 123.23, but a garbage value instead

10.5 ASSIGNING VALUES THROUGH A POINTER

You can use a pointer on the left-hand side of an assignment statement to assign a value to the location pointed to by the pointer. Assuming that p is an int pointer, this assigns the value 101 to the location pointed to by

```
p. *p = 101;
```

You can verbalize this assignment like this: “At the location pointed to by p, assign the value 101.” To increment or decrement the value at the location pointed to by a pointer, you can use a statement like this:

```
(*p)++;
```

The parentheses are necessary because the * operator has lower precedence than does the ++ operator.

The following program demonstrates an assignment through a pointer:

```

#include<iostream>
using namespace std;

int main()
{
    int *p, num;
    P=&num;

```

```

    *p=100;
    cout<<num<< ' \n;
    (*p)++;
    cout<<num<< ' \n;
    (*p)--;
    cout<<num<< ' \n;
    return 0;
}

```

The output from the program is shown here:

```

100
101
100

```

10.6 POINTERS AND ARRAYS

In C++, there is a close relationship between pointers and arrays. In fact, frequently a pointer and an array are interchangeable. Consider this fragment:

```
char str[80]; char *p1; p1 = str;
```

Here, str is an array of 80 characters and p1 is a character pointer. However, it is the third line that is of interest. In this line, p1 is assigned the address of the first element in the str array. (That is, after the assignment, p1 will point to str[0].) Here's why: In C++, using the name of an array without an index generates a pointer to the first element in the array. Thus, the assignment

```
p1 = str;
```

assigns the address of str[0] to p1. This is a crucial point to understand: When an unindexed array name is used in an expression, it yields a pointer to the first element in the array. Since, after the assignment, p1 points to the beginning of str, you can use p1 to access elements in the array. For example, if you want to access the fifth element in str, you can use

```
str[4]          or
```

```
*(p1+4)
```

Both statements obtain the fifth element. Remember, array indices start at zero, so when str is indexed, a 4 is used to access the fifth element. A 4 is also added to the pointer p1 to get the fifth element, because p1 currently points to the first element of str.

The parentheses surrounding `p1+4` are necessary because the `*` operation has a higher priority than the `+` operation. Without them, the expression would first find the value pointed to by `p1` (the first location in the array) and then add 4 to it. In effect, C++ allows two methods of accessing array elements: pointer arithmetic and array indexing. This is important because pointer arithmetic can sometimes be faster than array indexing—especially when you are accessing an array in strictly sequential order. Since speed is often a consideration in programming, the use of pointers to access array elements is very common in C++ programs. Also, you can sometimes write tighter code by using pointers instead of array indexing.

Here is an example that demonstrates the difference between using array indexing and pointer arithmetic to access the elements of an array. We will create two versions of a program that reverse the case of letters within a string. The first version uses array indexing. The second uses pointer arithmetic. The first version is shown here:

```
// Reverse case using array indexing.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    int i;
    char str[80] = "This Is A Test";
    cout << "Original string: " << str << "\n";
    for(i = 0; str[i]; i++)
    {
        if(isupper(str[i])) str[i] = tolower(str[i]);
        else if(islower(str[i])) str[i] = toupper(str[i]);
    }
    cout << "Inverted-case string: " << str;
    return 0
}
```

The output from the program is shown here:

```
Original string: This Is A Test Inverted-case string:
tHIS iS a tEST
```

Notice that the program uses the `isupper()` and `islower()` library functions to determine the case of a letter. The `isupper()` function returns true when its argument is an uppercase letter; `islower()` returns true when its argument is a lowercase letter. Inside the for loop, `str` is indexed, and the case of each letter is

checked and changed. The loop iterates until the null terminating str is indexed. Since a null is zero (false), the loop stops.

10.7 ARRAYS OF POINTERS

Pointers can be arrayed like any other data type. For example, the declaration for an int pointer array of size 10 is

```
int *pi[10];
```

Here, pi is an array of ten integer pointers. To assign the address of an int variable called var to the third element of the pointer array, you would write

```
int var;  
pi[2] = &var;
```

Remember, pi is an array of int pointers. The only thing that the array elements can hold are the addresses of integer values—not the values themselves. To find the value of var, you would write `*pi[2]`

Like other arrays, arrays of pointers can be initialized. A common use for initialized pointer arrays is to hold pointers to strings. Here is an example that uses a two-dimensional array of character pointers to implement a small dictionary:

```
#include<iostream>
#include<cstring>
using namespace std

int main()
{
    char *dictionary[ ][2]={“pencil”, “ A writing instrument”,
                             “keyboard”, “ An input device.”,
                             “rifle”,      “A      shoulder-fired
firearm”,
                             “airplane”,    “A      fixed-wing
aircraft.”,
                             “network”, “An interconnected
group of computers.”,
                             “ ”, “ ”
                             };
    char word[80];
    int l;
    cout<< “Enter word.”;
    cin>>word;

    for(i=0; *dictionary[i][0];i++)
```



```

    {
        if(!strcmp(dictionary[i][0], word);
        {
            cout<<dictionary[i][1]<< "\n";
            break;
        }
    }

    if(!*dictionary[i][0]
    cout<<word<< "not found. \n"
    return 0;
}

```

Here is a sample run:

```

Enter word: network
An interconnected group of computers.

```

When the array dictionary is created, it is initialized with a set of words and their meanings. Recall, C++ stores all string constants in the string table associated with your program, so the array need only store pointers to the strings. The program works by testing the word entered by the user against the strings stored in the dictionary. If a match is found, the meaning is displayed. If no match is found, an error message is printed.

Notice that dictionary ends with two null strings. These mark the end of the array. Recall that a null string contains only the terminating null character. The for loop runs until the first character in a string is null. This condition is tested with this expression:

```
*dictionary[i][0]
```

The array indices specify a pointer to a string. The * obtains the character at that location. If this character is null, then the expression is false and the loop terminates. Otherwise, the expression is true and the loop continues.

10.8 FUNCTION POINTER

Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to

Since a function pointer is nothing else than a variable, it must be defined as usual. In the following example

we define two function pointers named `pt2Function` and `pt2Member`. They point to functions, which take one float and two char and return an int. In the C++ example it is assumed, that the function, our pointer points to, is a member function of `TMyClass`.

```
int (*pt2Function) (float, char, char); // C
int (TMyClass::*pt2Member)(float, char, char); // C++
```

10.9 ASSIGN AN ADDRESS TO A FUNCTION POINTER

It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. It's optional to use the address operator `&` in front of the function's name.

Note: You may have got to use the complete name of the member function including class-name and scope-operator (`::`). Also you have got to ensure, that you are allowed to access the function right in scope where your assignment stands.

```
int Dolt (float a, char b, char c)
{
    printf("Dolt\n");
    return a+b+c;
}
int DoMore(float a, char b, char c)
{
    printf("DoMore\n");
    return a-b+c;
}
pt2Function = DoMore; // assignment
pt2Function = &Dolt; // alternative using address
operator

class TMyClass
{
public:
    int Dolt (float a, char b, char c)
    { cout << "TMyClass::Dolt" << endl; return
a+b+c; },
    int DoMore(float a, char b, char c)
```

```

    {
        cout << "TMyClass::DoMore" << endl;
        return a-b+c;
    };
    /* more of TMyClass */
};

pt2Member = TMyClass::Dolt; // assignment
pt2Member = &TMyClass::DoMore; // alternative
using address operator

```

10.10 CALLING A FUNCTION USING A FUNCTION POINTER

In C you have two alternatives of how to call a function using a function pointer: You can just use the name of the function pointer instead of the name of the function or you can explicitly dereference it. In C++ it's a little bit tricky since you need to have an instance of a class to call one of their (non-static) member functions. If the call takes place within another member function you can use the **this-pointer**.

Example :

```

int result1 = pt2Function (12, 'a', 'b'); // C short way
int result2 = (*pt2Function) (12, 'a', 'b'); // C
TMyClass instance;
int result3 = (instance.*pt2Member)(12, 'a', 'b'); // C++
int result4 = (*this.*pt2Member)(12, 'a', 'b'); // C++ if
this-pointer can be used

```

10.11 PASS A FUNCTION POINTER AS AN ARGUMENT

You can pass a function pointer as a function's calling argument. You need this for example if you want to pass a pointer to a callback function. The following code shows how to pass a pointer to a function which returns an int and takes a float and two char:

// <pt2Func> is a pointer to a function which returns an int and takes a float and two char

```

void PassPtr(int (*pt2Func)(float, char, char))
{
    float result = pt2Func(12, 'a', 'b'); // call using function
    pointer
    cout << result << endl;
}

```

```
}
```

// execute example code - 'Dolt' is a suitable function like defined above in 2.1-4

```
void Pass_A_Function_Pointer()
{
    cout << endl << "Executing
'Pass_A_Function_Pointer'" << endl;
    PassPtr(&Dolt);
}
```

10.12 RETURN A FUNCTION POINTER

It's a little bit tricky but a function pointer can be a function's return value. In the following example there are two solutions of how to return a pointer to a function which is taking a float and returns two float. If you want to return a pointer to a member function you have just got to change the definitions/declarations of all function pointers.

```
// function takes a char and returns a pointer to a function
float (*GetPtr1(const char opCode))(float, float)
{
    if(opCode == '+') return &Plus;
    if(opCode == '-') return &Minus;
}
```

// solution using a typedef

```
typedef float(*pt2Func)(float, float);
```

```
// <opCode> specifies which function to return
pt2Func GetPtr2(const char opCode)
```

```
{
    if(opCode == '+') return &Plus;
    if(opCode == '-') return &Minus;
}
```

// execute example code

```
void Return_A_Function_Pointer()
{
    cout << endl << "Executing
'Return_A_Function_Pointer'" << endl;
```

```

        float (*pt2Function)(float, float); // define a function
        pointer
        pt2Function=GetPtr1('+'); // get function pointer from
        function 'GetPtr1'
        cout << pt2Function(2, 4) << endl; // call function
        using the pointer
        pt2Function=GetPtr2('-'); // get function pointer from
        function 'GetPtr2'
        cout << pt2Function(2, 4) << endl; // call function
        using the pointer
    }

```

10.13 POINTERS TO OBJECTS

You can access an object either directly (as has been the case in all preceding examples), or by using a pointer to that object. To access a specific element of an object when using a pointer to the object, you must use the arrow operator: `->`. It is formed by using the minus sign followed by a greater-than sign.

To declare an object pointer, you use the same declaration syntax that you would use to declare a pointer for any other type of data.

The next program creates a simple class called `P_example`, defines an object of that class called `ob`, and defines a pointer to an object of type `P_example` called `p`. It then illustrates how to access `ob` directly, and how to use a pointer to access it indirectly

```

#include<iostream>
using namespace std;

class P_example
{
    int num;
    public:
    void set_num(int val)    {Num=val;}
    void show_num(){cout<<num<< "\n";
};

int main()
{
    P_example ob, *p;

    ob.set_num(1);    //Call functions directly on ob
    ob.show_num();

```

```

        P=&ob;        // Assign P the address of ob
        P->set_num(20); //Call functions through a pointer
to tob
        P->show_num();

        return 0;
    }

```

10.14 OPERATORS NEW AND NEW[]

In order to request dynamic memory we use the operator `new`. `new` is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

```

pointer          =          new          type
pointer = new type [number_of_elements]

```

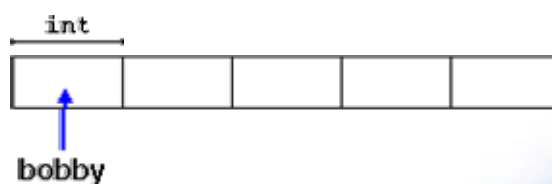
The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to assign a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:

```

int * bobby;
bobby = new int [5];

```

In this case, the system dynamically assigns space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `bobby`. Therefore, now, `bobby` points to a valid block of memory with space for five elements of type `int`.



The first element pointed by `bobby` can be accessed either with the expression `bobby[0]` or the expression `*bobby`. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with `bobby[1]` or `*(bobby+1)` and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an

array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type `bad_alloc` is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the default method used by `new`, and is the one used in a declaration like:

```
bobby = new int [5]; // if it fails an exception is thrown
```

The other method is known as `nothrow`, and what happens when it is used is that when a memory allocation fails, instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a null pointer, and the program continues its execution.

This method can be specified by using a special object called `nothrow` as parameter for `new`:

```
bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if `bobby` took a null pointer value:

```
int * bobby;  
bobby = new (nothrow) int [5];  
if (bobby == 0)  
{  
    // error assigning memory. Take measures.  
};
```


This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

10.15 OPERATOR DELETE AND DELETE[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
#include <iostream>
using namespace std;

int main ()
{
    int i,n;
    int * p;
    cout << "How many numbers would you like to
type? ";
    cin >> i;
    p= new (nothrow) int[i];
    if (p == 0)
        cout << "Error: memory could not be allocated";
    else
    {
        for (n=0; n<i; n++)
        {
            cout << "Enter number: ";
            cin >> p[n];
        }
    }
}
```

```

        cout << "You have entered: ";
        for (n=0; n<i; n++)
            cout << p[n] << ", ";
        delete[] p;
    }
    return 0;
}

```

The output is:

```

How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,

```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.



VIRTUAL FUNCTION

Unit Structure

1. Virtual Base Classes
2. Abstract Classes
3. Virtual Functions
4. Pure Virtual Functions

11.1 VIRTUAL BASE CLASSES

This ambiguity can be resolved if the class derived contains only one copy of the class base. This can be done by making the base class a virtual class. This keyword makes the two classes share a single copy of their base class . It can be done as follows :

```
class base
{
:
:
};

class Aclass : virtual public base
{
:
:
};

class Bclass : virtual public base
{
:
:
};

class derived : public Aclass, public Bclass
{
:
:
};
```

This will resolve the ambiguity involved.

2. ABSTRACT CLASSES

Abstract classes are the classes, which are written just to act as base classes. Consider the following classes.

```
class base
{
:
};
class Aclass : public base
{
:
:
};
class Bclass : public base
{
:
};
class Cclass : public base
{
:
:
};
void main()
{
    Aclass objA;
    Bclass objB;
    Cclass objC;
    :
    :
}
```

There are three classes – Aclass, Bclass, Cclass – each of which is derived from the class base. The main () function declares three objects of each of these three classes. However, it does not declare any object of the class base. This class is a general class whose sole purpose is to serve as a base class for the other three. Classes used only for the purpose of deriving other classes from them are called as abstract classes. They simply serve as base class , and no objects for such classes are created.

3. VIRTUAL FUNCTIONS

The keyword virtual was earlier used to resolve ambiguity for a class derived from two classes, both having a common ancestor. These classes are called virtual base classes. This time it helps in implementing the idea of polymorphism with class inheritance. The

function of the base class can be declared with the keyword `virtual`. The program with this change and its output is given below.

```
class Shape
{
    public :
    virtual void print()
    {
        cout << " I am a Shape " << endl;
    }
};
class Triangle : public Shape
{
    public :
    void print()
    {
        cout << " I am a Triangle " << endl;
    }
};
class Circle : public Shape
{
    public :
    void print()
    {
        cout << " I am a Circle " << endl;
    }
};

void main()
{
    Shape S;
    Triangle T;
    Circle C;

    S.print();
    T.print();
    C.print();

    Shape *ptr;
    ptr = &S;
    ptr -> print();
    ptr = &T;
    ptr -> print();
    ptr = &C;
    ptr -> print();
}
```

The output of the program is given below:

```
I am a Shape
I am a Triangle
I am a Circle
I am a Shape
I am a Triangle
I am a Circle
```

Now, the output of the derived classes are invoked correctly. When declared with the keyword `virtual`, the compiler selects the function to be invoked, based upon the contents of the pointer and not the type of the pointer. This facility can be very effectively used when many such classes are derived from one base class. Member functions of each of these can be then, invoked using a pointer to the base class.

11.4 PURE VIRTUAL FUNCTIONS

As discussed earlier, an abstract class is one, which is used just for deriving some other classes. No object of this class is declared and used in the program. Similarly, there are pure virtual functions which themselves won't be used. Consider the above example with some changes.

```
class Shape
{
    public :
    virtual void print() = 0; // Pure virtual function
};

class Triangle : public Shape
{
    public :
    void print()
    {
        cout << " I am a Triangle " << endl;
    }
};

class Circle : public Shape
{
    public :
    void print()
    {
        cout << " I am a Circle " << endl;
    }
};

void main()
```



```

{
    Shape S;
    Triangle T;
    Circle C;
    Shape *ptr;
    ptr = &T;
    ptr -> print();
    ptr = &C;
    ptr -> print();
}

```

The output of the program is given below:

```

I am a Triangle
I am a Circle

```

It can be seen from the above example that , the print() function from the base class is not invoked at all . even though the function is not necessary, it cannot be avoided, because , the pointer of the class Shape must point to its members. Object oriented programming has altered the program design process. Exciting OOP concepts like polymorphism have given a big boost to all this. Inheritance has further enhanced the language. This session has covered some of the finer aspects of inheritance. The next session will resolve some finer aspects of the language.

EXAMPLE:-

// virtual members

```

#include <iostream.h>
class CPolygon
{
    protected:
    int width, height;
    public:
    void set_values (int a, int b)
    {
        width=a; height=b;
    }
    virtual int area (void)
    {
        return (0);
    }
};
class CRectangle: public CPolygon

```

```

{
    public:
    int area (void)
    {
        return (width * height);
    }
};

class CTriangle: public CPolygon
{
    public:
    int area (void)
    {
        return (width * height / 2);
    }
};

int main ()
{
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;

    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;

    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);

    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
    return 0;
}

```

The output is :

```

20
10
0

```

Now the three classes (CPolygon, CRectangle and CTriangle) have the same members: width, height, set_values() and area(). area() has been defined as virtual because it is later redefined in derived classes. You can verify if you want that if you remove this word (virtual) from the code and then you execute the program the result will be 0 for the three polygons instead of 20,10,0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called for all of them since the calls are via a pointer to CPolygon.

Therefore what the word virtual does is to allow that a member of a derived class with the same name as one in the base class be suitably called when a pointer to it is used, as in the above example. Note that in spite of its virtuality we have also been able to declare an object of type CPolygon and to call its area() function, that always returns 0 as the result.



STREAMS AND FILES

Unit Structure

1. Input Output With Files
 2. Methods of Input and Output Classes
 3. Text mode files
 4. State flags
 5. get and put stream pointers
 6. Binary files
- 12.8 I/O Manipulators

12.1 INPUT OUTPUT WITH FILES

The techniques for file input and output, i/o, in C++ are virtually identical to those introduced in earlier lessons for writing and reading to the standard output devices, the screen and keyboard. To perform file input and output the include file `fstream` must be used.

```
#include <fstream>
```

`Fstream` contains class definitions for classes used in file i/o. Within a program needing file i/o, for each output file required, an object of class `ofstream` is instantiated. For each input file required, an object of class `ifstream` is instantiated. The `ofstream` object is used exactly as the `cout` object for standard output is used. The `ifstream` object is used exactly as the `cin` object for standard input is used. This is best understood by studying an example.

C++ has support both for input and output with files through the following classes:

`ofstream`: File class for writing operations (derived from `ostream`)

`ifstream` : File class for reading operations (derived from `istream`)

`fstream` : File class for both reading and writing operations (derived from `iostream`)

Open a file

The first operation generally done on an object of one of these classes is to associate it to a real file, that is to say, to open a file. The open file is represented within the program by a stream object (an instantiation of one of these classes) and any input or output performed on this stream object will be applied to the physical file.

In order to open a file with a stream object we use its member function `open()`:

```
void open (const char * filename, openmode mode);
```

where `filename` is a string of characters representing the name of the file to be opened and `mode` is a combination of the following flags:

```
ios::in Open file for reading
ios::out Open file for writing
ios::ate Initial position: end of file
ios::app Every output is appended at the end of file
ios::trunc If the file already existed it is erased
ios::binary Binary mode
```

These flags can be combined using bitwise operator OR: `|`. For example, if we want to open the file "example.bin" in binary mode to add data we could do it by the following call to function-member **open**:

```
ofstream file;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

All of the member functions `open` of classes `ofstream`, `ifstream` and `fstream` include a default mode when opening files that varies from one to the other:

class	default mode to parameter
<code>ofstream</code>	<code>ios::out ios::trunc</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

The default value is only applied if the function is called without specifying a mode parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined.

Since the first task that is performed on an object of classes **ofstream**, **ifstream** and **fstream** is frequently to open a file, these three classes include a constructor that directly calls the **open**

member function and has the same parameters as this. This way, we could also have declared the previous object and conducted the same opening operation just by writing:

```
ofstream file ("example.bin", ios::out | ios::app | ios::binary);
```

Both forms to open a file are valid.

You can check if a file has been correctly opened by calling the member function **is_open()**:

```
bool is_open();
```

that returns a **bool** type value indicating **true** in case that indeed the object has been correctly associated with an open file or **false** otherwise.

Closing a file

When reading, writing or consulting operations on a file are complete we must close it so that it becomes available again. In order to do that we shall call the member function **close()**, that is in charge of flushing the buffers and closing the file. Its form is quite simple:

```
void close ();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function **close**.

12.2 METHODS OF INPUT AND OUTPUT CLASSES

The **ifstream** class has several useful methods for input. These method are also in the class **cin**, which is used to read from standard input. These methods are used to read from any input stream. An input stream is a source of input such as the keyboard, a file or a buffer.

1) Extraction Operator >>

This overloaded operator handles all built in C++ data types. By default, any intervening white space is disregarded. That is, blanks, tabs, new lines, formfeeds and carriage returns are skipped over.

2) get()

This form of get extracts a single character from the input stream, that is, from the standard input, a file or a buffer. It does not skip white space. It returns type int.

3) get(char &ch)

This form of get also extracts a single character from the input stream, but it stores the value in the character variable passed in as an argument.

4) get(char *buff, int buffsize, char delimiter='\n')

This form of get reads characters into the C-style buffer passed as an argument buffsize characters are read, the delimiter is encountered or an end of file is encountered. The '\n' is the new line character. The delimiter is not read into the buffer but is instead left in the input stream. It must be removed separately but using either another get or an ignore. Because of this added step, this form of get is a frequent source of errors and should be avoided. Fortunately, another method shown below, getline, reads in the delimiter as well and should be used in place of this form of get.

5) Getline

There are several useful forms of getline.

ignore(int count=1, int delim=traits_type::eof)

This method reads and discards "count" characters from the input stream.

6) peek()

This method returns the next character from the input stream, but does not remove it. It is useful to look ahead at what the next character read will be.

7) putback(char &ch)

This method puts ch onto the input stream. The character in ch will then be the next character read from the input stream.

8) unget()

This method puts the last read character back into the input stream.

9) read(char *buff, int count)

This method is used to perform an unformatted read of count bytes from the input stream into a character buffer.

The ofstream class has several useful methods for writing to an output stream. An output stream is standard output (usually the screen), a file or a buffer. These methods are also in the object

cout, which is used for standard output. The simplest way to understand how to use these methods is by looking at a few examples. Since we have seen the extraction, >>, and insertion, << in several lessons, let's look at the other methods. Getline, which is very useful to read entire lines of text into a string.

Suppose we need to read a file and determine the number of alphanumeric characters, the number of blanks and the number of sentences. To determine the number of sentences we will count the number of periods (dots). We will disregard newlines and tabs.

Here is a program that solves the problem.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    int blank_count = 0;
    int char_count = 0;
    int sentence_count = 0;
    char ch;
    ifstream iFile("c:/lesson12.txt");
    if (!iFile)
    {
        cout << "Error opening input file" << endl;
        return -1;
    }
    while (iFile.get(ch))
    {
        switch (ch)
        {
            case ' ':
                blank_count++;
                break;

            case '\n':
            case '\t':
                break;

            case '.':
                sentence_count++;
                break;

            default:
                char_count++;
                break;
        }
    }
}
```

```

    }
    cout << "There are " << blank_count << " blanks" <<
endl;
    cout << "There are " << char_count << " characters"
<< endl;
    cout << "There are " << sentence_count << "
sentences" << endl;
    return 0;
}

```

As a second example, let's implement a program that will copy the contents of one file to another. The program will prompt the user for the input and output file names, and then copy.

```

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    char ch;
    string iFileName;
    string oFileName;
    cout << "Enter the source file name: ";
    cin >> iFileName;
    cout << "Enter the destination file name: ";
    cin >> oFileName;
    ofstream oFile(oFileName.c_str());
    ifstream iFile(iFileName.c_str());
    //Error checking on file opens omitted for brevity.
    while (iFile.get(ch))
    {
        oFile.put(ch);
    }
    return 0;
}

```

12.3 TEXT MODE FILES

Classes ofstream, ifstream and fstream are derived from ostream, istream and iostream respectively. That's why fstream objects can use the members of these parent classes to access data.

Generally, when using text files we shall use the same members of these classes that we used in communication with the console (cin and cout).

As in the following example, where we use the overloaded insertion operator <<:

```
// writing on a text file
#include <fstream.h>
int main ()
{
    ofstream examplefile ("example.txt");
    if (examplefile.is_open())
    {
        examplefile << "This is a line.\n";
        examplefile << "This is another line.\n";
        examplefile.close();
    }
    return 0;
}
```

The output is : This is a line.
 This is another line.

Data input from file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main ()
{
    char buffer[256];
    ifstream examplefile ("example.txt");
    if (! examplefile.is_open())
    { cout << "Error opening file"; exit (1); }
    while (! examplefile.eof() )
    {
        examplefile.getline (buffer,100);
        cout << buffer << endl;
    }
    return 0;
}
```

The output is : This is a line.
 This is another line.

This last example reads a text file and prints out its content on the screen. Notice how

we have used a new member function, called `eof` that `ifstream` inherits from class `ios` and that returns true in case that the end of the file has been reached.

12.4 STATE FLAGS

In addition to `eof()`, other member functions exist to verify the state of the stream (all of them return a bool value):

- 1) **`bad()`** : Returns true if a failure occurs in a reading or writing operation. For example in case we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- 2) **`fail()`** : Returns true in the same cases as `bad()` plus in case that a format error happens, as trying to read an integer number and an alphabetical character is received.
- 3) **`eof()`** : Returns true if a file opened for reading has reached the end.
- 4) **`good()`** : It is the most generic: returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by the previous member functions you can use member function `clear()`, with no parameters.

12.5 GET AND PUT STREAM POINTERS

All i/o streams objects have, at least, one stream pointer:

- 1) **`ifstream`**, like **`istream`**, has a pointer known as get pointer that points to the next element to be read.
- 2) **`ofstream`**, like **`ostream`**, has a pointer put pointer that points to the location where the next element has to be written.
- 3) **`fstream`**, like **`iostream`**, inherits both: get and put

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

`tellg()` and `tellp()`

These two member functions admit no parameters and return a value of type `pos_type` (according ANSI-C++ standard) that

is an integer data type representing the current position of get stream pointer (in case of `tellg`) or put stream pointer (in case of `tellp`).

`seekg()` and `seekp()`

This pair of functions serve respectively to change the position of stream pointers `get` and `put`. Both functions are overloaded with two different prototypes:

```
seekg ( pos_type position );
seekp ( pos_type position );
```

Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions `tellg` and `tellp`.

```
seekg ( off_type offset, seekdir direction );
seekp ( off_type offset, seekdir direction );
```

Using this prototype, an offset from a concrete point determined by parameter `direction` can be specified. It can be:

```
ios::beg Offset specified from the beginning of the stream
ios::cur Offset specified from the current position of the
stream pointer
ios::end Offset specified from the end of the stream
```

The values of both stream pointers `get` and `put` are counted in different ways for text files than for binary files, since in text mode files some modifications to the appearance of some special characters can occur. For that reason it is advisable to use only the first prototype of **`seekg`** and **`seekp`** with files opened in text mode and always use nonmodified values returned by **`tellg`** or **`tellp`**. With binary files, you can freely use all the implementations for these functions. They should not have any unexpected behavior.

The following example uses the member functions just seen to obtain the size of a binary file:

```
// obtaining file size
#include <iostream.h>
#include <fstream.h>
const char * filename = "example.txt";

int main ()
{
    long l,m;
    ifstream file (filename, ios::in|ios::binary);
    l = file.tellg();
```



```

        file.seekg (0, ios::end);
        m = file.tellg();
        file.close();
        cout << "size of " << filename;
        cout << " is " << (m-l) << " bytes.\n";
        return 0;
    }

```

The output is : size of example.txt is 40 bytes.

12.6 BINARY FILES

In binary files inputting and outputting data with operators like << and >> and functions like **getline**, does not make too much sense, although they are perfectly valid. File streams include two member functions specially designed for input and output of data sequentially: **write** and **read**. The first one (**write**) is a member function of **ostream**, also inherited by **ofstream**. And **read** is member function of **istream** and it is inherited by **ifstream**. Objects of class **fstream** have both. Their prototypes are:

```

write ( char * buffer, streamsize size );
read ( char * buffer, streamsize size );

```

Where buffer is the address of a memory block where the read data are stored or from where the data to be written are taken. The size parameter is an integer value that specifies the number of characters to be read/written from/to the buffer.

```

// reading binary file
#include <iostream.h>
#include <fstream.h>
const char * filename = "example.txt";

int main ()
{
    char * buffer;
    long size;
    ifstream file (filename,
        ios::in|ios::binary|ios::ate);
    size = file.tellg();
    file.seekg (0, ios::beg);
    buffer = new char [size];

```

```

        file.read (buffer, size);
        file.close();
        cout << "the complete file is in a buffer";
        delete[] buffer;
        return 0;
    }

```

The output is : the complete file is in a buffer

7. BUFFERS AND SYNCHRONIZATION

When we operate with file streams, these are associated to a buffer of type `streambuf`.

This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an out stream, each time the member function `put` (write a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in the buffer for that stream.

When the buffer is flushed, all data that it contains is written to the physical media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:

When the file is closed: before closing a file all buffers that have not yet been completely written or read are synchronized.

When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.

Explicitly with manipulators: When certain manipulators are used on streams synchronization takes place. These manipulators are: `flush` and `endl`.

Explicitly with function `sync()`: Calling member function `sync()` (no parameters) causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated buffer or in case of failure.

8. I/O MANIPULATORS

Up till now, we have accepted the default output formatting. C++ defines a set of manipulators which are used to modify the

state of iostream objects. These control how data is formatted. They are defined in the include file, `<ios>`. It is not usually necessary to explicitly include this file because it is included indirectly via the use of other includes such as `<iostream>` or `<fstream>`.

Let's see how some of these manipulators work in a simple program.

Manipulator	Use
Boolalpha	Causes bool variables to be output as true or false.
Noboolalpha (default)	Causes bool variables to be displayed as 0 or 1
Dec (default)	Specifies that integers are displayed in base 10.
Hex	Specifies that integers are displayed in hexadecimal.
Oct	Specifies that integers are displayed in octal.
Left	Causes text to be left justified in the output field.
Right	Causes text to be right justified in the output field.
Internal	Causes the sign of a number to be left justified and the value to be right justified.
Noshowbase (default)	Turns off displaying a prefix indicating the base of a number.
Showbase	Turns on displaying a prefix indicating the base of a number.
Noshowpoint (default)	Displays decimal point only if a fractional part exists.
Showpoint	Displays decimal point always.
noshowpos (default)	No "+" prefixing a positive number.
Showpos	Displays a "+" prefixing a positive number.
Skipws (default)	Causes white space (blanks, tabs, newlines) to be skipped by the input operator, <code>>></code> .
Noskipws	White space not skipped by the extraction operator, <code>>></code> .
Fixed (default)	Causes floating point numbers to be displayed in fixed notation.

Scientific	Causes floating point numbers to be displayed in scientific notation.
Nouppercase (default)	0x displayed for hexadecimal numbers, e for scientific notation
Uppercase	0X displayed for hexadecimal numbers, E for scientific notation

The manipulators in the above table modify the state of the iostream object. This means that once used on an iostream object they will effect all subsequent input or output done with the object. There are several other manipulators that are used to format a particular output but do no modify the state of the object.

Setting Output Width

setw(w) - sets output or input width to w; requires <iomanip> to be included.

width(w) - a member function of the iostream classes.

Filling White Space

setfill(ch) - fills white space in output fields with ch; requires <iomanip> to be included.

fill(ch) = a member function of the iostream classes.

Setting Precision

setprecision(n) - sets the display of floating point numbers at precision n. This does not effect the way floating point numbers are handled during calculations in your program.

Here is a simple program illustrating the use of the i/o manipulators.

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
    int intValue = 15;
    cout << "Integer Number" << endl;
    cout << "Default: " << intValue << endl;
    cout << "Octal: " << oct << intValue << endl;
```

```

    cout << "Hex: " << hex << intValue << endl;
    cout << "Turning showbase on" << showbase <<
endl;
    cout << "Dec: " << dec << intValue << endl;
    cout << "Octal: " << oct << intValue << endl;
    cout << "Hex: " << hex << intValue << endl;
    cout << "Turning showbase off" << noshowbase <<
endl;
    cout << endl;
    double doubleVal = 12.345678;
    cout << "Floating Point Number" << endl;
    cout << "Default: " << doubleVal << endl;
    cout << setprecision(10);
    cout << "Precision of 10: " << doubleVal << endl;
    cout << scientific << "Scientific Notation: " <<
doubleVal << endl;
    cout << uppercase;
    cout << "Uppercase: " << doubleVal << endl;
    cout << endl;
    bool theBool = true;
    cout << "Boolean" << endl;
    cout << "Default: " << theBool << endl;
    cout << boolalpha << "BoolAlpha set: " << theBool <<
endl;
    cout << endl;
    string myName = "John";
    cout << "Strings" << endl;
    cout << "Default: " << myName << endl;
    cout << setw(35) << right << "With setw(35) and right:
"<< myName << endl;
    cout.width(20);
    cout << "With width(20): " << myName << endl;
    cout << endl;
    return 0;
}

```



TEMPLATES

Unit Structure :

- 13.1 Function templates
- 13.2 Class templates

13.1 FUNCTION TEMPLATES

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;  
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword `class` or the keyword `typename`. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>  
myType GetMax (myType a, myType b)  
{  
    return (a>b?a:b);  
}
```

Here we have created a template function with `myType` as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the

function template `GetMax` returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call `GetMax` to compare two integer values of type `int` we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of `myType` by the type passed as the actual template parameter (`int` in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b)
{
    T result;
    result = (a>b)? a : b;
    return (result);
}

int main ()
{
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

The output is:

```
6
10
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b)
{
    return (a>b?a:b);
}
int main ()
{
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax(i,j);
    n=GetMax(l,m);
    cout << k << endl;
```

```

        cout << n << endl;
        return 0;
    }

```

The output is :

```

        6
       10

```

Notice how in this case, we called our function template `GetMax()` without explicitly specifying the type between angle-brackets `<>`. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class `T`) and the function template itself accepts two parameters, both of this `T` type, we cannot call our function template with two objects of different types as arguments:

```

    int i;
    long l;
    k = GetMax (i,l);

```

This would not be correct, since our `GetMax` function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```

template <class T, class U>
T GetMin (T a, U b)
{
    return (a<b?a:b);
}

```

In this case, our function template `GetMin()` accepts two parameters of different types and returns an object of the same type as the first parameter (`T`) that is passed. For example, after that declaration we could call `GetMin()` with:

```

    int i,j;
    long l;
    i = GetMin<int,long> (j,l);

```

or simply:

```

    i = GetMin (j,l);

```

even though `j` and `l` have different types, since the compiler can determine the appropriate instantiation anyway.

13.2 CLASS TEMPLATES

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair
{
    T values [2];
public:
    mypair (T first, T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the `template <...>` prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair
{
    T a, b;
public:
    mypair (T first, T second)
    {
```

```

        a=first;
        b=second;
    }
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main ()
{
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}

```

The output is:

100

Notice the syntax of the definition of member function getmax:

```

template <class T>
T mypair<T>::getmax ()

```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.



EXCEPTIONS

Unit Structure:

- 14.1 Exceptions
- 14.2 Exception specifications

14.1 EXCEPTIONS

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
// exceptions
#include <iostream>
using namespace std;

int main ()
{
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "An exception occurred. Exception  Nr.
" << e << endl;
    }
    return 0;
}
```


EXCEPTIONS

Unit Structure:

- 14.1 Exceptions
- 14.2 Exception specifications

14.1 EXCEPTIONS

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
// exceptions
#include <iostream>
using namespace std;

int main ()
{
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "An exception occurred. Exception  Nr.
" << e << endl;
    }
    return 0;
}
```

The output is :

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try
{
    // code here
}
catch (int param)
{
    cout << "int exception";
}
catch (char param)
{
    cout << "char exception";
}
catch (...)
{
    cout << "default exception";
}
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
try
{
    try
    {
        // code here
    }
    catch (int n)
    {
        throw;
    }
}

catch (...)
{
    cout << "Exception occurred";
}
```

14.2 EXCEPTION SPECIFICATIONS

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called `myfunction` which takes one argument of type `char` and returns an element of type `float`. The only exception that this function might throw is an exception of type `int`. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular `int`-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw(); // no exceptions allowed
int myfunction (int param);         // all exceptions allowed
```

14.3 STANDARD EXCEPTIONS

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `exception` and is defined in the `<exception>` header file under the namespace `std`. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called `what` that returns a null-terminated character sequence (`char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

Example

```
// standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }
} myex;

int main ()
{
    try
    {
        throw myex;
    }

    catch (exception& e)
    {
        cout << e.what() << endl;
    }
    return 0;
}
```

The output is :

My exception happened.

We have placed a handler that catches exception objects by reference (notice the ampersand `&` after the type), therefore this catches also classes derived from `exception`, like our `myex` object of class `myexception`.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `std::exception` class. These are:

Exception	Description
<code>bad_alloc</code>	thrown by <code>new</code> on allocation failure
<code>bad_cast</code>	thrown by <code>dynamic_cast</code> when fails with a referenced type
<code>bad_exception</code>	thrown when an exception type doesn't match any catch
<code>bad_typeid</code>	thrown by <code>typeid</code>
<code>ios_base::failure</code>	thrown by functions in the <code>iostream</code> library

For example, if we use the operator `new` and the memory cannot be allocated, an exception of type `bad_alloc` is thrown:

```
try
{
    int * myarray= new int[1000];
}
catch (bad_alloc&)
{
    cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a `bad_alloc` exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a `bad_alloc` exception.

Because `bad_alloc` is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

```
// bad_alloc standard exception
#include <iostream>

#include <exception>
using namespace std;
```

```
int main ()
{
    try
    {
        int* myarray= new int[1000];
    }
    catch (exception& e)
    {
        cout << "Standard exception: " << e.what() <<
endl;
    }
    return 0;
}
```



THE C++ STANDARD TEMPLATE LIBRARY

Unit Structure :

- 15.1 Introduction
- 15.2 Standard Templates
- 15.3 Stacks
- 15.4 Older Stacks
- 15.5 Queues
- 15.6 Older Queues
- 15.7 Lists
- 15.8 Vectors
- 15.9 Iterators and Containers
- 15.10 Algorithms

1. INTRODUCTION

This is a short primer on the **STL** or "Standard Template Library" for the programming language C++ as defined in the 1997 standard. The **STL** is a collection C++ libraries that allow you to use several well known kinds of data structures with out having to program them. They are designed so that the code runs efficiently. The compiler does most of the work of generating the efficient implementations. The libraries include a large number of possibilities. we describe some of things you can do with the STL versions of stacks, queues, vectors, and lists. I define some important and useful ideas in the **Glossary** below. The is a table summarizing the methods used with stacks, queues, vectors and lists at **Summary** below. For each kind of data structure I give working examples of how to declare, access and use objects of that type.

2. STANDARD TEMPLATES

The international and American Standard for C++ requires there to be special libraries that implement the following generic data structures.

Library	Description
<vector>	A dynamic array [\$Vectors]
<list>	A randomly changing sequence of items
<stack>	A sequence of items with pop and push at one end only
<queue>	A Sequence of items with pop and push at opposite ends
<deque>	Double Ended Queue with pop and push at both ends
<bitset>	A subset of of a fixed and small set of items
<set>	An unordered collection of items
<map>	An collection of pairs of items indexed by the first one

These are designed to be general, efficient, and powerful rather than easy to use.

. (end of section **Standard Templates**) <<Contents | End>>

15.3 STACKS

Stacks are only accessed at their top. To be able to use **STL** stacks in a file of C++ source code or a header file add

```
#include <stack>
```

at the beginning of the file.

Suppose that *T* is any type or class - say an int, a float, a struct, or a class, then

```
stack<T> s;
```

declares a new and empty stack called *s*. Given *s* you can:

- 1) test to see if it is empty:
 s.empty()
- 2) find how many items are in it:
 s.size()
- 3) push a *t* of type *T* onto the top:
 s.push(t)
- 4) pop the top off *s*:
 s.pop()

- 5) get the top item of s
 s.top()
 6) change the top item:
 s.top() = expression.

Example of an STL Stack

```
void reverse(string & x)
//Afterwards x will have the same elements in the opposite
order.
{
    stack<char> s;
    const int n = x.length();
    //Put characters from x onto the stack
    for(int i=0; i<n; ++i)
        s.push(x[i]);
    //take characters off of stack and put them back into x
    for(int i=0; !s.empty(); ++i, s.pop())
        x[i]=s.top();
}
```

15.4 OLDER STACKS

On some older C++ libraries you are forced to indicate how the stack is implemented whenever you declare one. You write either

```
stack< vector<T> > s;
```

or

```
stack< list<T> > s;
```

Example :

```
void reverse(string & x) //INOUT: x is a string of
characters
//Afterwards x will have the same elements in the
opposite order.
{
    stack< vector<char> > s;
    const int n = x.length();
    for(int i=0; i<n; ++i)
        s.push(x[i]);
    for(int i=0; !s.empty(); ++i, s.pop())
        x[i]=s.top();
}
..... ( end of section Stacks) <<Contents | End>>
```

5. QUEUES

Queues allow data to be added at one end and taken out of the other end. To be able to use **STL** queues add this before you start using them in your source code:

```
#include <queue>
```

Suppose that T is any type or class - say an int, a float, a struct, or a class, then

1) declares a new and empty queue called q . Given an object q :

```
queue<T> q;
```

2) test to see if q is empty:

```
q.empty()
```

3) find how many items are in q :

```
q.size()
```

4) push a $t:T$ onto the end of q :

```
q.push(t)
```

5) pop the front of q off q :

```
q.pop()
```

6) get the front item of q :

```
q.front()
```

7) change the front item:

```
q.front() = expression.
```

8) get the back item of q :

```
q.back()
```

9) change the back item:

```
q.back() = expression.
```

Example of putting three items into a queue and then taking them off the queue.

```
#include <iostream.h>
#include <list>
#include <queue>
int main()
{
```

```

queue<char> q;
q.push('a');
q.push('b');
q.push('c');
cout << q.front();
q.pop();
cout << q.front();
q.pop();
cout << q.front();
q.pop();
}

```

15.6 OLDER QUEUES

On some older C++ libraries you are forced to indicate how the queue is implemented whenever you declare one. You write

```
queue< list<T> > q;
```

in place of

```
queue< T > q;
```

Example on putting three items into a queue and then taking them off the queue.

```

#include <iostream.h>
#include <list>
#include <queue>
int main()
{
    queue< list <char> >q;
    q.push('a');
    q.push('b');
    q.push('c');
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();
}
..... ( end of section Queues) <<Contents | End>>

```

15.7 LISTS

Lists are good when data needs to be reorganized a lot. To be able to use **STL** lists add this before you start using them in your source code:

```
#include <list>
```

Suppose that T is any type or class - say an int, a float, a struct, or a class, then

1) To declare a new and empty list called l . Given an object t : `list<T> l;`

2) test to see if l is empty:

```
l.empty()
```

3) find how many items are in l :

```
l.size()
```

4) push a $t:T$ onto the end of l :

```
l.push_back(t)
```

5) pop the last off l :

```
l.pop_back()
```

6) push a $t:T$ onto the start of l :

```
l.push_front(t)
```

7) pop the front of l off l :

```
l.pop_front()
```

8) get the front item of l :

```
l.front()
```

9) change the front item:

```
l.front() = expression.
```

10) get the back item of l :

```
l.back()
```

11) change the back item:

```
l.back() = expression.
```

12) Sort the list:

```
l.sort()
```

13) Clear the list:

```
l.clear()
```

14) Merge in a sorted list into a sorted list:

```
l.merge(list_of_sorted_elements)
```


15) Reverse the list:

```
l.reverse()
```

16) Assign a copy of *q1* to *q*:

```
q = q1
```

[Lists and Iterators]

Example of List Processing

This uses a utility function 'print' that is implemented below [**An Example of Iterating Through a List**]

Building and Sorting a List

//Using a list to sort a sequence of 9 numbers.

```
#include <iostream.h>
```

```
#include <list>
```

```
// #include <algorithm>
```

```
void print(list<int> a); //utility function print lists of ints
```

```
int main()
```

```
{
```

```
    list<int> a;
```

```
    //Put 9,8,7,6,5,4,3,2,1 onto the list
```

```
    for(int i=0; i<9;++i)
```

```
        a.push_back(9-i); // put new element after all the others
```

```
    print(a); //here the list contains (9,8,7,6,5,4,3,2,1)
```

```
    a.sort(); //in the <list> library!
```

```
    print(a); //here the list contains (1,2,3,4,5,6,7,8,9)
```

```
}
```

```
..... ( end of section Lists) <<Contents | End>>
```

15.8 VECTORS

Vectors are good when we have an unknown sequence of items to store but we want to access the by their sequence numbers. To be able to use **STL** vectors add this before you start using them in your source code:

```
#include <vector>
```

Suppose that *T* is any type or class - say an int, a float, a struct, or a class, then

1) to declares a new and empty **vector** called *v*. Given object *v* declare like the above:

```
vector<T> v;
```

- 2) test to see if v is empty:
 `v.empty()`
- 3) find how many items are in v :
 `v.size()`
- 4) push a $t:T$ onto the end of v :
 `v.push_back(t)`
- 5) pop the front of v off v :
 `v.pop_back()`
- 6) get the front item of v :
 `v.front()`
- 7) change the front item:
 `v.front() = expression.`
- 8) get the back item of v :
 `v.back()`
- 9) change the back item:
 `v.back() = expression.`
- 10) Access the i th item ($0 \leq i < \text{size}()$) without checking to see if it exists:
 `v[i]`
- 11) Access the i th item safely:
 `v.at(i)`
- 12) Assign a copy of $v1$ to v :
 `v = v1`

Sorting and operating on all items efficiently, see **Iterators** below.
 (end of section **Vectors**) <<Contents | End>>

15.9 ITERATORS AND CONTAINERS

Containers:

A Container is a data structure that holds a number of object of the same type or class. The oldest example of a Container in C++ is an array. Even in C you had arrays and you would typically write code like this:

```

const int MAX = 10;
float a[MAX];
...
for(int i=0; i<10; ++i)
    process(a[i]);
...

```

Lists, Vectors, Stacks, Queues, etc are all **Containers**.

C arrays were designed so that they could be accessed by using a variable that stores the address of an item. These variables are called *pointers*. They are used like this with an array:

```

for(float *p=a; p!=p+MAX; ++p)
    process(*p);
...

```

Iterators:

Items in **Containers** are referred to be special objects called: *iterators*. They are generalization of C's pointers. With an iterator class *Iter* you can process each item in a vector or a list by similar code:

```

for( Iter p=c.begin(); p!=c.end(); ++p)
    process(*p);

```

All **Containers** C in the STL have a number of iterator objects. Container class C has an iterator called C::iterator

For any type T, list<T> and vector<T> are **Containers**. So there are iterator classes called list<T>::iterator and they are used like this:

```

for( list<T>::iterator p=c.begin(); p!=c.end(); ++p)
    process(*p);

```

Vector iterators have type

vector<T>::iterator

and used like this:

```

for( vector<T>::iterator p=c.begin(); p!=c.end(); ++p)
    process(*p);

```

If you change your choice from a **vector** to a **list** then the code is almost identical. This makes your code easier to modify.

For any **container** class C of objects type T and any object c of type C:

Declare an iterator for **container** of type C.

`C::iterator p`

Move iterator *p* onto the next object in *c*(if any!).

`++p`

The value selected by *p*.

`*p`

The iterator that refers to the first item in *c*

`c.begin()`

The iterator that refers to one beyond *c*.

`c.end()`

Declare an iterator for **container** of type C and set to the start of *c*.

`C::iterator p = c.begin();`

Test to see if iterator *p* has come to the end of object *c*:

`p != c.end();`

assign *p1* to *p* -- afterwards both refer to same item

`p = p1;`

Lists and Iterators

Insertion and deletion of items at the start or inside a List of elements is controlled by an iterator:

Insert item *x* into List / before iterator *p*

`l.insert(p, x);`

Erase the element pointed at by iterator *q* in List /

`l.erase(q);`

Example:

```
#include <iostream.h>

#include <list>
void print(list <char> );// elsewhere
main()
{
    list <char> l;
    list <char>::iterator p;
    l.push_back('o');
    l.push_back('a');
    l.push_back('t');
    p=l.begin();
```

```

('o', 'a', 't')
cout << " "<< *p<<endl; // p refers to the 'o' in
                           cout << " " << *p<<endl;
                           print(l);
                           l.insert(p, 'c');
                           // l is now ('c', 'o', 'a', 't') and p still refers to 'o'
                           cout << " "<< *p<<endl;
                           print(l);
                           l.erase(p);
                           cout << " "<< *p<<endl; // p refers to an 'o' but it
is not in l!
                           print(l);
                           l.erase(l.begin()); //removes front of l
                           print(l);
                           }

```

Ranges:

A pair of iterators describes a **range** of items in their **container**. The items in the **range** start with the first iterator in the pair. The **range** has all the following items up to just before the item referred to by the last iterator of the pair.

Suppose that *first* and *last* form a **range** and *it* is an iterator then:
for(it=first; it != last; it++)

will refer to each element in the **range** [*first..last*) in turn. In the body of the for loop the value of the element is **it*.

Given a **container** *c*, then *c.begin()* and *c.end()* form a **range**.

An Example of Iterating Through a List

```

void print( list<int> a)
{
    for(list<int>::iterator ai=a.begin(); ai!=a.end(); ++ai)
        cout << *ai << " ";
    cout << endl;
    cout << "-----"<<endl;
}

```

15.10 ALGORITHMS

Ranges and Iterators are used several useful functions and algorithms in the **STL**. Suppose you have a type or class of data called *T* and in a program are working with a vector *v* of type *vector<T>* then

`vector<T>::iterator`

is the class of suitable iterators. Here are two useful values:

```

v.begin()
v.end()

```

These always form a **range** of all the items in *v* from the **front** up to and including the **back**. You can sort a **vector** *v* simply by writing:

```
sort(c.begin(), c.end());
```

Example : Sorting a vector with 9 integers

```
#include <iostream.h>
#include <vector>
#include <algorithm>
void print( vector<int> ) ;//utility function outputs a $container
int main()
{
    vector<int> a;
    // Place 9,8,7,6,5,4,3,2,1 into the vector
    for(int i=0; i<9;++i)
        a.push_back(9-i);// put new element after all the others
    print(a); // elements of a are (9,8,7,6,5,4,3,2,1)
    sort( a.begin(), a.end() ); //in the STL <algorithm> library
    print(a); // elements are now in order.
}
void print( vector<int> a)
{
    for(vector<int>::iterator ai=a.begin(); ai!=a.end(); ++ai)
        cout << *ai << " ";
    cout << endl;
    cout << "-----"<<endl;
}
..... ( end of section Iterators) <<Contents | End>>
```

Summary

Templat e	push/pop	Commo n	items	Extras
stack	pop(),push(T)	empty(), size()	top()	-
queue	pop(),push(T)	empty(), size()	front(),back())	-
vector	push_back(T , pop_back()	empty(), size()	front(), back()	[int], at(int), =
list	push_back(), pop_back(), push_front(T , pop_front()	empty(), size()	front(), back()	sort(), clear(), reverse(), merge(l),at(int , =

Note:

You must always `#include` headers if you use the words: `vector`, `list`, `string`, `queue`, or `stack` in your program or in a header file.

There must be a space between the two ">" signs below:

```
stack< vector< T > > s;
```

If the standard `<list>` and `<vector>` is not found then you are using an older C++ compiler.

On some older compilers and current libraries when you need a `<string>` as well as either `<list>` or `<vector>` you need to `#include <string>`

before including the `list` or `vector` rather in the reverse order! Older string library appear to define some special versions of `vector` and `list` operators and the older compilers can not make up its mind which to use.



EXERCISE**Exercise : 1**

1. It has been said that C++ sits at the center of the modern programming universe. Explain this statement.
2. A C++ compiler produces object code that is directly executed by the computer. True or false?
3. What are the three main principles of object-oriented programming?
4. Where do C++ programs begin execution?
5. What is a header?
6. What is `<iostream>`? What does the following code do?
`#include <iostream>`
7. What is a namespace?
8. What is a variable?
9. Which of the following variable names is/are invalid?
a. count b. _count c. count27 d. 67count e. if
10. How do you create a single-line comment? How do you create a multiline comment?
11. Show the general form of the if statement. Show the general form of the for loop.
12. How do you create a block of code?
13. The moon's gravity is about 17 percent that of Earth's. Write a program that displays a table that shows Earth pounds and their equivalent moon weight. Have the table run from 1 to 100 pounds. Output a newline every 25 pounds.
14. A year on Jupiter (the time it takes for Jupiter to make one full circuit around the Sun) takes about 12 Earth years. Write a program that converts Jovian years to Earth years. Have the user specify the number of Jovian years. Allow fractional years.
15. When a function is called, what happens to program control?

16. Write a program that averages the absolute value of five values entered by the user. Display the result.

Exercise: 2

1. What type of integers are supported by C++?
2. By default, what type is 12.2?
3. What values can a bool variable have?
4. What is the long integer data type?
5. What escape sequence produces a tab? What escape sequence rings the bell?
6. A string is surrounded by double quotes. True or false?
7. What are the hexadecimal digits?
8. Show the general form for initializing a variable when it is declared.
9. What does the % do? Can it be used on floating-point values?
10. Explain the difference between the prefix and postfix forms of the increment operator.
11. Which of the following are logical operators in C++?
a. && b. ## c. || d. \$\$ e. !
12. How can `x = x + 12;` be rewritten?
13. What is a cast?
14. Write a program that finds all of the prime numbers between 1 and 100.

Exercise: 3

1. Write a program that reads characters from the keyboard until a \$ is typed. Have the program count the number of periods. Report the total at the end of the program.
2. In the switch, can the code sequence from one case run into the next? Explain.

3. Show the general form of the if-else-if ladder.
4. Show the for statement for a loop that counts from 1000 to 0 by – 2.
5. Explain what break does.
6. In the following fragment, after the break statement executes, what is displayed?
7. The increment expression in a for loop need not always alter the loop control variable by a fixed amount. Instead, the loop control variable can change in any arbitrary way. Using this concept, write a program that uses a for loop to generate and display the progression 1, 2, 4, 8, 16, 32, and so on.
8. The ASCII lowercase letters are separated from the uppercase letters by 32. Thus, to convert a lowercase letter to uppercase, subtract 32 from it. Use this information to write a program that reads characters from the keyboard. Have it convert all lowercase letters to uppercase, and all uppercase letters to lowercase, displaying the result. Make no changes to any other character. Have the program stop when the user enters a period. At the end, have the program display the number of case changes that have taken place.
9. What is C++'s unconditional jump statement?

Exercise: 4

1. Show how to declare a short int array called hightemps that is 31 elements long.
2. In C++, all arrays begin indexing at _____.
3. Write a program that searches an array of ten integers for duplicate values. Have the program display each duplicate found.
4. What is a null-terminated string?
5. Write a program that prompts the user for two strings and then compares the strings for equality, but ignores case differences. Thus, "ok" and "OK" will compare as equal.
6. When using strcat(), how large must the recipient array be?
7. In a multidimensional array, how is each index specified?

8. Show how to initialize an int array called nums with the values 5, 66, and 88.
9. What is the principal advantage of an unsized array declaration?
10. What is a pointer? What are the two pointer operators?
11. Can a pointer be indexed like an array? Can an array be accessed through a pointer?
12. Write a program that counts the uppercase letters in a string. Have it display the result.
13. What is it called when one pointer points to another pointer?
14. Of what significance is a null pointer in C++?

Exercise : 5

1. Show the general form of a function.
2. Create a function called `hypot()` that computes the length of the hypotenuse of a right triangle given the lengths of the two opposing sides. Demonstrate its use in a program. For this problem, you will need to use the `sqrt()` standard library function, which returns the square root of its argument. It has this prototype:
`double sqrt(double val);` It uses the header `<cmath>`.
3. Can a function return a pointer? Can a function return an array?
4. Create your own version of the standard library function `strlen()`. Call your version `mystrlen()`, and demonstrate its use in a program.
5. Does a local variable maintain its value between calls to the function in which it is declared?
6. Give one benefit of global variables. Give one disadvantage.
7. Create a function called `byThrees()` that returns a series of numbers, with each value 3 greater than the preceding one. Have the series start at 0. Thus, the first five numbers returned by `byThrees()` are 0, 3, 6, 9, and 12. Create another function called `reset()` that causes `byThrees()` to start the series over again from 0. Demonstrate your functions in a program. Hint: You will need to use a global variable.
8. Write a program that requires a password that is specified on the command line. Your program doesn't have to actually do anything

except report whether the password was entered correctly or incorrectly.

9. A prototype prevents a function from being called with the improper number of arguments. True or false?

10. Write a recursive function that prints the numbers 1 through 10. Demonstrate its use in a program.

Exercise : 6

1. What are the two ways that an argument can be passed to a subroutine?

2. In C++, what is a reference? How is a reference parameter created?

3. Given this fragment,

```
int f(char &c, int *i); // ... char ch = 'x'; int i = 10; show how to call f( )
with the ch and i.
```

4. Create a void function called `round()` that rounds the value of its double argument to the nearest whole value. Have `round()` use a reference parameter and return the rounded result in this parameter. You can assume that all values to be rounded are positive. Demonstrate `round()` in a program. To solve this problem, you will need to use the `modf()` standard library function, which is shown here:

```
double modf(double num, double *i);
```

The `modf()` function decomposes `num` into its integer and fractional parts. It returns the fractional portion and places the integer part in the variable pointed to by `i`. It requires the header `<cmath>`.

5. Modify the reference version of `swap()` so that in addition to exchanging the values of its arguments, it returns a reference to the smaller of its two arguments. Call this function `min_swap()`.

6. Why can't a function return a reference to a local variable?

7. How must the parameter lists of two overloaded functions differ?

8. In Project 6-1, you created a collection of `print()` and `println()` functions. To these functions, add a second parameter that specifies an indentation level. For example, when `print()` is called like this,

`print("test", 18);` output will indent 18 spaces and then will display the string "test". Have the indentation parameter default to 0 so that when it is not present, no indentation occurs.

9. Given this prototype,

`bool myfunc(char ch, int a=10, int b=20);` show the ways that `myfunc()` can be called.

10. Briefly explain how function overloading can introduce ambiguity.

Exercise : 7

1. Show how to declare an int variable called test that can't be changed by the program. Give it an initial value of 100.

2. The volatile specifier tells the compiler that a variable might be changed by forces outside the program. True or false?

3. In a multifile project, what specifier do you use to tell one file about a global variable declared in another file?

4. What is the most important attribute of a static local variable?

5. Write a program that contains a function called `counter()`, which simply counts how many times it is called. Have it return the current count.

6. How does `&` differ from `&&`?

7. What does this statement do? `x *= 10;`

8. Using the `rrotate()` and `lrotate()` functions from Project 7-1, it is possible to encode and decode a string. To code the string, left-rotate each letter by some amount that is specified by a key. To decode, right-rotate each character by the same amount. Use a key that consists of a string of characters. There are many ways to compute the number of rotations from the key. Be creative. The solution shown in the online answers is only one of many.

9. On your own, expand `show_binary()` so that it shows all bits within an unsigned int rather than just the first eight.

Exercise : 8

1. What is the difference between a class and an object?

2. What keyword is used to declare a class?

3. What does each object have its own copy of?
4. Show how to declare a class called Test that contains two private int variables called count and max.
5. What name does a constructor have? What name does a destructor have?
6. Given this class declaration:


```
class sample
{
    int i;
public:
    sample(int x) {i = x;}
    //..
};
```

 show how to declare a Sample object that initializes i to the value 10.
7. When a member function is declared within a class declaration, what optimization automatically takes place?
8. Create a class called Triangle that stores the length of the base and height of a right triangle in two private instance variables. Include a constructor that sets these values. Define two functions. The first is hypot(), which returns the length of the hypotenuse. The second is area(), which returns the area of the triangle.
9. Expand the Help class so that it stores an integer ID number that identifies each user of the class. Display the ID when a help object is destroyed. Return the ID when the function getID() is called.

Exercise: 9

1. What is a copy constructor and when is it called? Show the general form of a copy constructor.
2. Explain what happens when an object is returned by a function. Specifically, when is its destructor called?
3. Given this class:


```
class T {
    int i,j;
public:
    int sum(){return i+j;}
};
```

 show how to rewrite sum() so that it uses this.

4. What is a structure? What is a union?
5. Inside a member function, to what does `*this` refer?
6. What is a friend function?
7. Show the general form used for overloading a binary member operator function.
8. To allow operations involving a class type and a built-in type, what must you do?
9. Can the `?` be overloaded? Can you change the precedence of an operator?
10. For the Set class developed in Project 9-1, define `<` and `>` so that they determine if one set is a subset or a superset of another set. Have `<` return true if the left set is a subset of the set on the right, and false otherwise. Have `>` return true if the left set is a superset of the set on the right, and false otherwise.
11. For the Set class, define the `&` so that it yields the intersection of two sets.
12. On your own, try adding other Set operators. For example, try defining `|` so that it yields the symmetric difference between two sets. The symmetric difference consists of those elements that the two sets do not have in common.

Exercise: 10

1. A class that is inherited is called a _____ class. The class that does the inheriting is called a _____ class.
2. Does a base class have access to the members of a derived class? Does a derived class have access to the members of a base class?
3. Create a derived class of TwoDShape called Circle. Include an `area()` function that computes the area of the circle.
4. How do you prevent a derived class from having access to a member of a base class?
5. Show the general form of a constructor that calls a base class constructor.

6. Given the following hierarchy:

Class alpha{...

Class beta : public alpha{..

Class gamma : public beta{...

in what order are the constructors for these classes called when a Gamma object is instantiated?

7. How can protected members be accessed?

8. A base class pointer can refer to a derived class object. Explain why this is important as it relates to function overriding.

9. What is a pure virtual function? What is an abstract class?

10. Can an object of an abstract class be instantiated?

11. Explain how the pure virtual function helps implement the “one interface, multiple methods” aspect of polymorphism.

Exercise : 11

1. What are the four predefined streams called?

2. Does C++ define both 8-bit and wide-character streams?

3. Show the general form for overloading an inserter.

4. What does `ios::scientific` do?

5. What does `width()` do?

6. An I/O manipulator is used within an I/O expression. True or false?

7. Show how to open a file for reading text input.

8. Show how to open a file for writing text output.

9. What does `ios::binary` do?

10. When the end of the file is reached, the stream variable will evaluate as false. True or false?

11. Assuming a file is associated with an input stream called `strm`, show how to read to the end of the file.

12. Write a program that copies a file. Allow the user to specify the name of the input and output file on the command line. Make sure that your program can copy both text and binary files.

13. Write a program that merges two text files. Have the user specify the names of the two files on the command line in the order they should appear in the output file. Also, have the user specify the name of the output file. Thus, if the program is called `merge`, then the following command line will merge the files `MyFile1.txt` and `MyFile2.txt` into `Target.txt`: `merge MyFile1.txt MyFile2.txt Target.txt`

14. Show how the `seekg()` statement will seek to the 300th byte in a stream called `MyStrm`.

Exercise : 12

1. Explain how `try`, `catch`, and `throw` work together to support exception handling.

2. How must the catch list be organized when catching exceptions of both base and derived classes?

3. Show how to specify that a `MyExcpt` exception can be thrown out of a function called `func()` that returns `void`.

4. Define an exception for the generic `Queue` class shown in Project 12-1. Have `Queue` throw this exception when an overflow or underflow occurs. Demonstrate its use.

5. What is a generic function, and what keyword is used to create one?

6. Create generic versions of the `quicksort()` and `qs()` functions shown in Project 5-1. Demonstrate their use.

7. Using the `Sample` class shown here, create a queue of three `Sample` objects using the generic `Queue` shown in Project 12-1:

```
class sample
{
    int id;
public:
    sample()
    { id =0; }
    sample(int x){id=x;}
    void show(){cout<<id<<endl;}
};
```

8. Rework your answer to question 7 so that the `Sample` objects stored in the queue are dynamically allocated.

9. Show how to declare a namespace called `RobotMotion`.

10. What namespace contains the C++ standard library?

11. Can a static member function access the non-static data of a class?
12. What operator obtains the type of an object at runtime?
13. To determine the validity of a polymorphic cast at runtime, what casting operator do you use?
14. What does `const_cast` do?
15. On your own, try putting the Queue class from Project 12-1 in its own namespace called `QueueCode`, and into its own file called `Queue.cpp`. Then rework the `main()` function so that it uses a `using` statement to bring `QueueCode` into view.
16. Continue to learn about C++. It is the most powerful computer language currently available. Mastering it puts you in an elite league of programmers.

